O'REILLY

TURING 图灵程序设计丛书



代码之外的功夫

程序员精进之路

Programming Beyond Practices

人工智能时代不被AI取代的编程职业规划路径

[美] Gregory T. Brown 著

李志 译

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

译者介绍

李志

1995年生于山东济南。目前在西安交通大学人工智能与机器人研究所从事计算机视觉相关研究，同时在西安交通大学软件学院攻读软件工程硕士学位。本科毕业于西安交通大学外国语学院英语系，英语专业八级，同时具备英语语言文学功底和计算机专业知识。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵程序设计丛书

代码之外的功夫：程序员精进之路

Programming Beyond Practices: Be More Than Just a Code Monkey

［美］Gregory T. Brown 著
李志 译

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北京

图书在版编目（CIP）数据

代码之外的功夫 ： 程序员精进之路 / （美） 格雷戈里·布朗 (Gregory T. Brown) 著 ； 李志译. -- 北京 ： 人民邮电出版社, 2018.3
（图灵程序设计丛书）
ISBN 978-7-115-47837-5

Ⅰ. ①代… Ⅱ. ①格… ②李… Ⅲ. ①程序设计 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2018)第017143号

内 容 提 要

本书虽然面向程序员，却不包含代码。在作者看来，90% 的程序设计工作都不需要写代码；程序员不只是编程专家，其核心竞争力是利用代码这一工具解决人类社会的常见问题。以此作为出发点，作者精心构思了 8 个故事，以情景代入的方式邀请读者思考代码之外的关键问题：软件开发工作如何从以技术为中心转为以人为本？透过故事主人公的视角，读者能比较自己与书中角色的差异，发现决策过程的瑕疵，提升解决问题的综合能力。

书中的故事涵盖程序员的整个软件开发生涯，但经过了浓缩，可供所有软件开发人员快速阅读。

◆ 著　　　　[美] Gregory T. Brown
译　　　　李 志
责任编辑　谢婷婷
执行编辑　回 春
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　　印刷

◆ 开本：800×1000　1/16
印张：7.75
字数：183千字　　　　　　2018年3月第1版
印数：1－3 500册　　　　　2018年3月北京第1次印刷

著作权合同登记号　图字：01-2017-6481号

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

版权声明

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些"细微的信号"来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的"动物书"；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，O'Reilly 的每一项产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

"O'Reilly Radar 博客有口皆碑。"

——*Wired*

"O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。"

——*Business 2.0*

"O'Reilly Conference 是聚集关键思想领袖的绝对典范。"

——*CRN*

"一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。"

——*Irish Times*

"Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：'如果你在路上遇到岔路口，走小路（岔路）。'回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。"

——*Linux Journal*

目录

前言

关于本书

本书不是教材，而是一本“故事集”。书中收录的各个小故事可以指导你正确地对待软件项目，并且更好地完成项目中的工作。

翻开本书，你不会找到任何教科书式的建议。相反，本书将为你呈现一系列案例。你将看到一线开发人员在实践中真正遇到的问题，以及寻找解决方案的过程。

为了增强代入感并鼓励你在阅读时发挥创造力，本书将你作为每个故事的主角。如果你感到有点怪怪的，这很正常。其实，我在写这篇介绍时也感觉挺怪的！

我希望这种情景代入方式可以让你学到更多，而不是觉得“这本书又是哪个编程专家口若悬河的说教”。我更希望你能提出这样的问题：“如果我置身于此情此景，真的会像书中那样做吗？如果不会，那是为什么呢？”

我建议你在阅读本书的过程中深入问题内部，从更深的层次反省自己的行为、习惯和看待问题的方式。最好在读书时手边能有本笔记，一有想法就赶紧记下来，以便日后与朋友和同事分享。本书中的概念需要经过思考和讨论，不能不求甚解地一带而过。

在每个故事里，你都会有不同的身份。每个情景都经过仔细的设计，能教给你有用的知识。但最重要的是，你要注意比较真实的自己和我为你设定的角色，寻找其中的共同点和值得深究的不同点。

没错，这样说好像有点急功近利了。但读书或写书的最终目的，不就是充实自己，让自己变得更好吗？我们都在朝着这个方向努力。有了你的配合和帮助，我觉得我们能够做得不错。

好了朋友，系好安全带，我们要出发了。

本书概览

本书的故事覆盖整个软件开发生涯，但经过了浓缩，可供所有一线开发人员快速阅读。

在第 1 章中，你是能力很强的程序员，并且正在发挥技术特长，运用速成原型法帮助人们探索新的产品创意。

在第 2 章中，工作变得复杂起来。你需要逐步扩展现有系统，同时需要为很多活跃客户提供支持。你发现理想丰满而现实骨感：一方面，你想按照自认为正确的想法进行工作；另一方面，客户不断给你施压，要求你尽快交付新的特性。

在第 3 章中，你对草率决策的代价，尤其是在整合外部代码的过程中仓促决策所带来的损失，有了更深的理解。你从过去的错误中学到很多，并开始关注业务、客户服务及技术工作三者之间的复杂关系。

在第 4 章中，你已经成为了经验相当丰富的开发人员，并且有能力教别人如何看待编程和解决问题。你现在已经开始指导新入行的朋友了。

在第 5 章中，你已经成为良师。你的开发经验已经非常丰富，足以单打独斗。即使是现场演示，你也能从容应对。你运用这些技能给学生上课，教他们怎样填补理论与实践之间的空白，将二者结合起来。

在第 6 章中，你在通往大师的道路上继续前行。你可以指出公司遗留系统的弱点，并设计合适的组件进行替换，既使商业效果得到优化，又使工作流程更加友好。

在第 7 章中，你对整个软件行业都已经足够熟悉。无论组织的哪个层级出现问题，你都能发现并修复。你的核心竞争力仍然是软件开发，但足够丰富的经验使你能很好地与各个层级的人员进行交流。

在第 8 章中，你开始思考整个计算机行业的发展问题。此刻，你可以自由选择自己未来的职业发展道路，所以现阶段最重要的问题变为：自己将如何发展，以及为何作此抉择？

由于软件开发人员的职业发展路线更像是螺旋线而非直线，因此无论你现在处于哪个阶段，我都建议你通读所有章节。

我写下的这些故事适用于不同水平的读者，本书也并不会给“初级”和“高级”划分界限。每一章都是独立的，所以也可以不按顺序阅读……但想要获得最好的效果，还是逐页阅读为好。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几百家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920047391.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

一本书就像一条辛苦织就的毯子，而作者只编织了其中的一部分。由于其自身的特殊性，本书尤其依赖他人的帮助和贡献，单靠我自己完全不可能完成这项工作。

首先，我要感谢你。本书的宗旨是为你服务，但依然要靠你自己挑起大梁。你花在本书上的时间和精力是我无法奢求的，我非常感谢这一点。

这部作品的完成需要感谢多位优秀的策划编辑：Jeff Bleiel、Brian MacDonald，以及 Mike Loukides。Mike 说服我编写本书，并自始至终都提供支持与反馈。Brian 帮助我度过了写作本书的早期阶段。Jeff 不知疲倦地和我一起工作，帮助我度过了写作项目最艰难的阶段：将潦草的手稿变为值得阅读的篇章。

我也非常幸运地拥有一支极其优秀的技术审校团队，其成员包括 Michael Feathers、Nell Shamrell、Saron Yitbarek 以及许多提供了有益反馈的“无名英雄”。针对作品的早期草稿，Ward Cunningham 也大致提供了一些富有见地的想法。

本书的文字编辑是 Stephanie Morillo，她将一堆杂乱的词句整理和润色成优美的语言，以飨读者。这本奇特的小书需要一个极其优秀的人来担当文字编辑，Stephanie 做得非常完美。

Kristen Brown 做事一丝不苟，并用细心和耐心主导了本书的出版过程。图书制作是我开始写本书时最担心的问题，但我们一开始合作，我的担忧就烟消云散了。

虽然本书很薄，但它是我和世界各地数百位慷慨的优秀人士之间千千万万次对话和共享经历的结晶。下面列出了其中的几十位，但这只是为本书提供过帮助的人中很小的一部分。

Ben Berkowitz、Sarah Bray、Florian Breisch、Melle Boersma、Ben Callaway、Christian Carter、Joseph Caudle、Mel Conway、Kenn Costales、Liam Dawson、Donovan Dikaio、Brad Ediger、Martin Fowler、Gregory Gibson、Melissa Gibson、Eric Gjersten、James Gifford、James Edward Gray II、David Haslem、Brian Hughes、Ron Jeffries、Alex Kashko、Kam Lasater、Tristan Lescut、Alexander Mankuta、Joseph McCormick、Steve Merrick、Alan Moore、Matthew Nelson、Carol Nichols、Calinoiu Alexandru Nicolae、Stephen Orr、Bruce Park、Srdjan Pejic、Vanja Radovanovic、Donald Reinertsen、Pito Salas、Clive Seebregts、Evan Sharp、Kathy Sierra、Derek Sivers、Danya Smith、Hunter Stevens、Jacob Tjørnholm、Gary Vaynerchuk、Jim Weirich、Solomon White、Jia Wu、Jan Žák……

其中几个人是以他们的工作成果间接地启发了我，但绝大部分人贡献出了他们宝贵的时间和我交流想法、讨论项目，启发了我在过去一年里研究和写作这个话题。

最后要强调的是，我要感谢 O’Reilly Media 所有为本书付出过劳动的员工，还要感谢 Tim O’Reilly，是他创建了这一如此优秀的出版公司。这本小书的出版过程奇特而又艰难，是所有人的竭力相助，才使本书最终得以问世。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

善用设计原型，探索项目创意

假设你在一家代理机构工作，负责指导客户做早期产品设计和项目规划。

不论做什么产品或项目，首要任务都是尽快发掘和实现客户头脑中的需求。在项目刚刚起步时，对话和绘制线框图是很有用的方法；但紧接着就应该进入探索式编程阶段，因为仅凭对话或画图能带给你的启发非常有限。

尽早生成可工作的软件，可以令产品设计变成交互式协作过程。高效的反馈环有利于快速识别潜在的不良设计，并对此提出解决方案，以免日后在更关键的阶段浪费大量时间和精力。

再简单的软件系统也是由很多活动的组件构成的，所以在设计的早期阶段就让它运转起来，看看组件之间的相互作用，是很有好处的。虽然所做的项目千差万别，但在这一层面上，所有项目都是相通的。

这一周，你将会和你的同伴 Samara 一起给一个音乐视频推荐系统开发功能原型。最初的特性集不需要打磨得多么完美，只要能实现基本功能，并从对产品感兴趣的人那里收集反馈即可。

本章主要内容

你将学到如何运用探索式编程技术，在开发过程开始后的几小时之内为产品创意构思出有意义的概念验证方案。

1.1 从理解项目背后的需求入手

对这个崭新的音乐视频推荐系统，你闻所未闻，所以不知道应该对它有何期待。于是，你约见了你的客户 Ross，想和他简单聊聊，以便了解从何处入手。

> **你**：Ross，你好！感谢你的接待。我的开发伙伴 Samara 也在听咱们的谈话。如果你方便的话，咱们随时可以开始。
>
> **Ross**：好的，我已经准备好了。第一步要做什么？
>
> **你**：是这样，我想先听一下你的想法。你为什么会对音乐视频推荐感兴趣呢？了解这一点可以帮助我们确定在原型设计过程中应该重点关注哪些问题。
>
> **Ross**：可以，没问题。我们开了一个博客，在里面发布一些精选音乐视频列表。这个博客到现在已经开了有好几年了。我们有专业的合作方，专攻各个种类的音乐列表制作。不过，人们如果想要查看某个列表，需要手动搜索。
>
> 几年来，我们在博客上分享了 4000 多个视频，已经形成一个相当大的音乐库，但现在唯一的检索方式仍然是查找博客文章。
>
> 我们现在开始考虑怎么能更便捷地检索音乐库。考虑了几种方案之后，我们认为构建一个推荐系统之类的东西也许行得通。
>
> 最初的版本可以比较简单，但最好能尽快做出点东西来。我们想先将它呈现给几十个最活跃的社区用户和博客内容贡献者。
>
> **你**：这个项目听起来很不错！那咱们开始干活吧！

了解了基本主题之后，你和 Ross 又聊了几分钟，大致探讨了概念验证方案。这类项目经常需要面对的一个问题是，新系统是要作为独立项目，还是要和现有的某些系统整合在一起。

Ross 自己也不是很明确自己的需求。你提出细节问题都可以暂时搁置，现在应该集中精力探讨这个想法究竟是否可行，他表示同意。

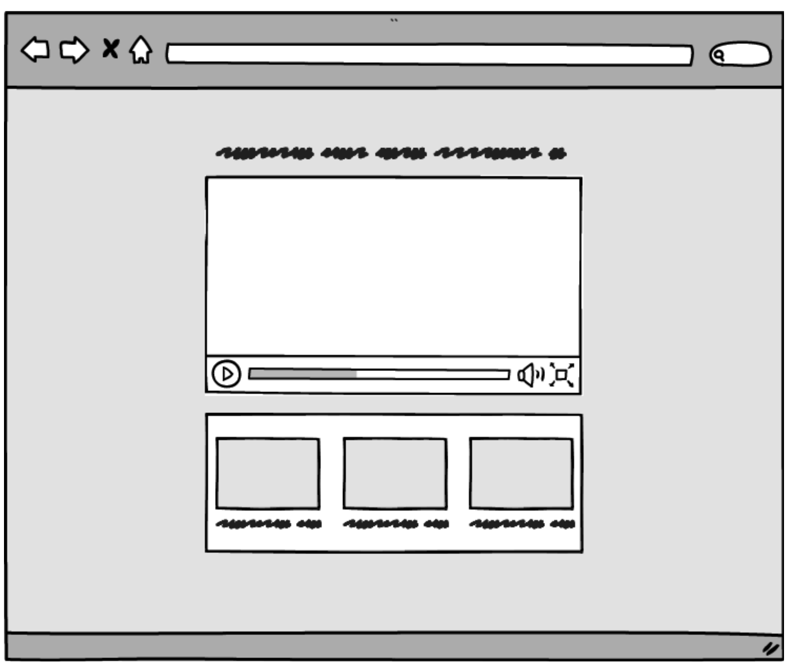
你们深入讨论了如何设计软件原型，以便让音乐视频博客的读者更容易上手。最终，你提出了一个简单易行的解决方案：用博客搜索视频，并将搜索结果作为原型中的样本。通过这种方法，新推荐系统中的内容对于 Ross 和博客读者来说都是熟悉的，而且新的应用和原来的网站之间会有很清晰的联系，即使二者在技术层面上是用完全不同的两套代码实现的。

1.2 利用线框图表达功能需求

弄清楚几个整体性问题之后，你把注意力转回到方法论上来：如何开始项目的第一轮开发。通常在这一阶段，绘制线框图非常有帮助。你可以通过线框图向大家解释待开发应用的基本结构，同时让大家都了解需要完成什么工作，以免因过度关注技术细节而迷失了大方向。

你建议不要花费过长时间去讨论推荐系统的实现方式，而应该先集中精力寻找“最简单可行的方法”。[1]

> **你**：刚开始设计用户界面时，我们可以从居中前置的视频播放器这个页面入手。在播放器下面可以放几个推荐视频的缩略图，这几个推荐视频是根据正在播放的视频选出来的。你觉得这个设计怎么样？
>
> Ross：不错，这样说好像挺有道理的。但我可能得见了实际效果才能知道具体情况。
>
> **你**：趁着我们聊天，Samara 一直在忙着画线框图。这张图应该能为我们的工作开个好头。稍等，我把图上传一下……



注 1：这一观点（http://pbpbook.com/wardc）由 Ward Cunningham 提出，目的在于提醒人们注意工作的最根本目标，不要因为过多地考虑收益损失而迷失方向。

你：你觉得如何？我们把问题尽量简化了，这样工作会比较好上手。

Ross：看起来还不错，跟我在别的网站上看到的视频播放器差不多。这对我们的用户来说应该挺容易理解的。

你：太好了！在我们继续讨论之前，我和 Samara 想根据这张图做一个真实的网页。所有的图片都将用占位图模拟，所以做这个网页不会花太长时间。我们要用这个模拟网页测试一些基本想法，好让余下的工作顺利进行。

Ross：没问题，如果你们觉得有帮助，就去做吧。

此刻你已经鼓足干劲，但是 Samara 好像有点茫然。你问她怎么回事，她解释说，就在你拿着她的线框图向 Ross 询问反馈的那一瞬间，她脑中灵光一闪，想到了一个更好的界面设计。

Samara 提出用一种新的播放器来代替原有的推荐方式。她表示可以在播放一个视频时引入“赞”和“踩”的按钮，这样观众可以表明自己的偏好；再加一个大大的“下一个视频”按钮，点击一下就能立即播放一个新推荐的视频。这有点像看电视时的换台操作，但智能系统可以预测出接下来观众可能想看的内容。

这个新想法非常不错，但是实现起来可能不太容易。在你们稍微权衡利弊之后，Samara 同意先实现前一个简单的想法，毕竟这样能更快将真实的网页交到真人手上测试使用。

1.3 编程之初立即搭建测试系统

速成原型法的意义是拉近项目中每个参与者之间的距离：不仅是开发人员和客户之间，也包括客户和用户之间。

要达到这些目标，搭建一个人人都能使用的交互系统很有必要。这样做不仅可以鼓励大家亲自操作和试验，而非纸上谈兵，还能让大家更方便地了解你的进度。有了这种想法，你便开始进入编写和发布 Web 应用的日常流程。

由于你使用了一个正规的应用托管平台，因此这种流程无非就是用你最喜欢的框架新建一个 Hello World 页面，然后把代码推送到一个支持你所用的工具链的 Git 仓库。从这一步开始，平台就可以自行安装所需依赖项，并自动启动 Web 服务器。

尽管这个 Web 应用对应的网址有些奇怪（类似于 baby-robot-pants-suit.somehostingprovider.com），但不出几分钟它就出现在真实的互联网环境里了。

在这个阶段，你对项目完成时生产环境最终的样子毫无概念，其实你也不太在乎。为了方便从客户的目标观众那里收集反馈，你会构建一些考察性的特性，这部分代码在产品完成上线之前就会被移除。

你把 Web 应用的基础结构部分中能砍掉的都砍掉了，甚至连数据库系统都暂时没有建，因为现在还不清楚是否需要。你玩的就是“大规模投资不足”，而且你还玩得不错。

“我们用不用给网页弄些自定义风格什么的？”Samara 问道。

你想了想，却想到了 YAGNI 原则，[2] 然后答案就明朗了。

你回答说：“不用。如果这个原型是为营销演示做的漂亮的软件，我们一开始就会重点关注它的外观。但对于目前的情形，我觉得 Ross 只是想把原型给几个朋友看看，让他们就功能方面提提意见。这样说来，因为它只是一个简单的视频播放应用，所以界面可以尽情精简，简到极致。”

这次 Samara 好像彻底服了，虽然你又一次“不修边幅”，删繁就简。不过你们一起工作的时间够长了，如今已经习惯了；很多情况下，Samara 也会为你指点迷津，防止你把事情想复杂。

你花几分钟搭建了一个你常用的 CSS 框架，同时 Samara 把一些占位图拼凑了一下。这些工作做完后，你写了一些简单的 HTML 代码，把图片和一些编好的标题对齐到网格。

你纠结于歌曲名称到底应该放哪儿、字体具体设多大，在这上面花的时间有点多。但是你马上想到了很重要的两点：第一，目前这些细节根本无关紧要；第二，该吃午饭了！

你直接按原样部署了代码。过了一分钟，网页就发布到网上了。



注 2：YAGNI（http://pbpbook.com/yagni）即 You Aren’t Gonna Need It（你不需要它）。这是一种设计原则，旨在告诉设计人员不要添加任何不必要的功能。

这个网页看起来实在是毫无特别之处。你开始担心客户理解不了为什么给他看这么简略的网页。

为了验证自己的想法，你问 Samara 怎么看。她指出，再简单的设计第一次见客户也免不了遭遇意见，但再丑的媳妇也得见公婆，这样说来晚见不如早见。

这下你放心了。你提醒自己，第一次发布的真正目的是创建一个可用的系统，以便提高后续的变更速度，并由此开始探索项目创意的过程。从现在开始，客户可以直接和你编写的软件交互，这将大大提高你的开发速度。

你给 Ross 发了一条消息，告诉他你做好了一个页面并请他过目。然后，你休息了一小会儿，等待 Ross 的回复。

1.4 全面探讨不足，改善追求实效

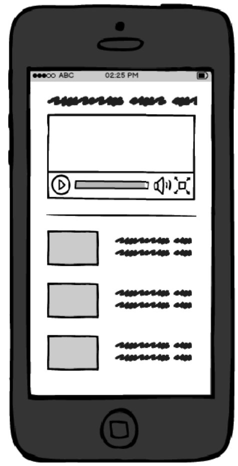
你回到办公桌前，看到 Ross 已经把反馈消息发过来了。

> 程序员朋友们好！
>
> 我刚刚试了一下这个页面。在我的笔记本电脑上，页面看起来和你们给我看的草图差不多，这没问题。
>
> 我又在手机上试了试，但页面看起来就有点别扭了。视频和屏幕一样宽，而且下面的推荐视频被排成长长的一列，而不是并列的一行。
>
> 我们绝不是希望刚一开始就让这个页面看起来多漂亮，但是至少可以让所有的推荐视频尽量在一屏里显示完整，而不是需要往下翻，去看一个又一个完整大小的视频。
>
> 你们觉得这个问题好解决吗？
>
> Ross

和 Samara 料想的一样，虽然第一次发布的页面已经简单到极致，但还是有问题浮出水面了。完全不犯错误是不可能的，但最关键的是你对这些问题作何反应。

你知道自己不能逃避手机界面的问题，但也不打算对它进行深究。对这个问题比较合理的回复是绘制一张新的线框图，然后解释一下正确渲染时页面应该是什么样的。

Samara 开始在自己的手机上浏览一些比较流行的视频网站，发现其中有好几个用了同样的布局。她随后画了一张差不多的草图，但只留下了其中最基本的元素。



你把线框图发给了 Ross，然后花了些时间跟他讨论接下来的几个步骤。

> Ross：谢谢你们的图！这可以说是一个很好的开始！
>
> **你**：很高兴听到你这样的评价。有件事需要我们现在决定一下：我们是马上修复手机布局的问题，还是以后再说？
>
> 我和 Samara 觉得还是先实现一些有用的推荐功能，再回头考虑界面问题比较好。
>
> 虽然这么说，但如果你认为即使是在收集反馈的阶段，手机界面的友好程度也很重要，我们可以在继续之前先解决这个问题。
>
> Ross：如果拖得越久，这种问题是不是越难解决？
>
> **你**：我觉得不会。推荐系统的大部分工作是在后台进行的，所以界面不需要改太多。而且如果我们真要对主界面做什么重大改变，手机界面还是需要推倒重制。
>
> Ross：那好，这个问题就暂缓吧。不过，如果早期的测试用户抱怨说这东西在手机上用起来不方便，我可能会改变主意。但是先不用担心，我们可以等到开始收集反馈时再说。

每当发现自己的软件有瑕疵时，你可能想要停下手中的工作，立即去修复。但是在项目的探索阶段，你得去平衡软件缺陷带来的损失和修复这一缺陷的时间成本。

在这种情况下，推迟修复手机界面小问题所节约下来的时间，可以用来研究音乐视频数据集，并尝试构思一些推荐规则。但这时你可以就修复问题提出一些粗略的计划，并和客户交流想法。这样一来，你就在没有投入太多精力的情况下提前排除了此问题可能带来的一些风险。

1.5 早问多问，验证设想

从之前的谈话中可以看出，Ross 只是想为自己的音乐社区做个好玩的东西，这比较容易实现。幸好他没有要求“给我做个全世界最高级的推荐型音乐播放服务出来”，也没有提出诸如此类的要求。

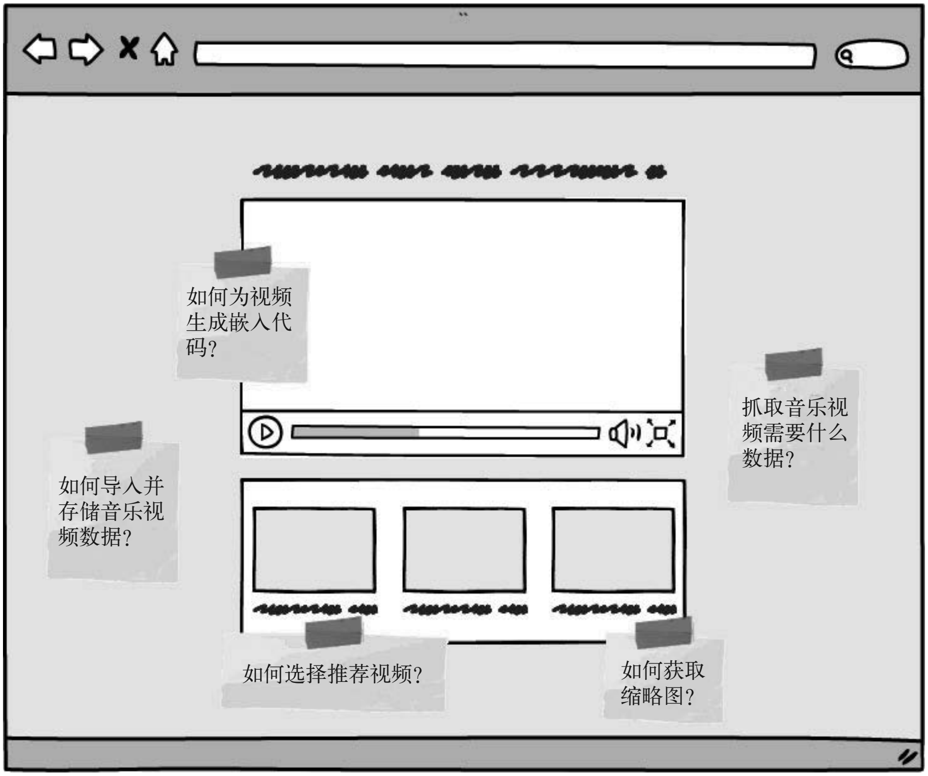
不过，验证对这种事情的设想总是有好处的，而且越早验证越好。最初的线框图主要关注用户界面的外观，而现在是时候讨论一下系统如何工作了。

> **你**：现在我想问一个技术性更强的问题……
>
> 我们在实现推荐功能时，应该采用什么规则呢？
>
> **Ross**：哦，这个嘛……我其实希望你们对此能有些见解。几周之前，我们甚至根本没想到这个项目值得去做，所以对它没什么研究。
>
> **你**：其实有很多方法可供选择，从最基本的类别匹配到非常高级的机器学习算法，不胜枚举。选择什么方法要看具体情况。虽然不管怎样我们都能帮助你做这个项目，但我们真的不太擅长选择推荐规则。
>
> **Ross**：我们的博客文章内容都是特别细化的精选列表（如“你可能从未听过的 10 首 Miles Davis 乐曲”“20 世纪 80 年代纽约嘻哈现场表演合集”“最受喜爱的圣诞季家庭音乐”），不知道这点有没有帮助。
>
> 我们希望要做的这个推荐工具能帮观众在众多列表之间快速切换，直接找到自己喜欢的音乐。比如，他们可能正在看一段 20 世纪 80 年代纽约嘻哈现场表演的视频，然后这个推荐工具就能找出同一位艺术家的其他作品，或者找出同一时代的其他嘻哈音乐作品，诸如此类。
>
> **你**：好的，这给了我们一些启发。谢谢你。我觉得我和 Samara 可能需要消化消化，思考思考，然后明天做出一些东西给你看看。你觉得可以吗？
>
> **Ross**：当然没问题！感谢你们所做的工作，非常有意思。

从这次对话中能够确定，一个简单的推荐系统应该就足够了，因为 Ross 在细节实现方面的需求很灵活。你这次算是走运，但如果下次客户头脑中有什么复杂的想法，早问总比晚问好。所谓“学之乃知，不问不识”，多问问总不会有坏处！

1.6 力求缩小自己的工作范围

从开始到现在所做的所有工作，还只是在为项目寻找一个切入点，而现在是时候撸起袖子加油干了。目前还有诸多未知因素亟待确定，你花几分钟研究了 Samara 之前画的草图，提出了很多关于细节实现的问题。



这些问题一股脑儿地涌入你的脑中，让你毫无头绪。但是如果不想办法为这些问题排出优先级，就没法进行下一步的工作。

在你和 Samara 发现的这 5 个重要问题中，有两个似乎比较容易解决：如何为视频生成嵌入代码，以及如何获取缩略图。

你找出了 Ross 管理的音乐博客，看看他用的视频托管平台是什么。你又点开了几篇博客文章，查看它们的源代码，研究其组织结构。

“大部分文章都直接嵌入了 FancyVideoService 里的视频。嵌入代码的格式是标准的，区别不同视频只需要依靠每个视频唯一的标识符。”

“那缩略图呢？是什么样的？”

你犹豫了一会儿，然后试着点击了博客里的几个静态的视频缩略图链接。由此确定，这是一些哑链接，点击它们并不会转到任何地址。

“这个网站好像根本没用真正意义上的缩略图。目前来看，所有内容都是来自别的网站的内嵌视频，所以估计我们得查一下。”

你在网上搜索了几分钟，想查查怎样抓取 FancyVideoService 视频的缩略图，但是并没有找到任何与此相关的官方文档。不过，你找到了一篇博客文章，里面描述了 FancyVideoService 使用的内部 URL 格式。因此，只需要从音乐博客的视频嵌入代码中获取视频的标识符，再套到这个格式里，就可以很容易地生成视频链接。

于是，你根据音乐博客里的视频，手动生成了一些缩略图 URL。它们看起来能正常工作，但无法确定在实际使用中这种做法是否行得通。现在你的态度比较乐观，但在项目收尾之前还得和 FancyVideoService 联系一下，确认这种做法是合法的。

扫除了这些障碍，现在你可以重新关注在审查草图时发现的一些更主观的问题：抓取音乐视频需要什么数据，如何存储这些数据，以及如何用这些数据生成有用的推荐信息。

你和 Samara 开始讨论可行的方法，但很快就发现你们对细节关注得过多了。所以，你们又回到了那个经典问题：“最简单可行的方法是什么？”

沉思了一小会儿后，Samara 突然灵光一闪。

“我们先从艺术家匹配做起，怎么样？根据正在播放的视频，我们随机抓取同一艺术家的几个其他作品。”

“好主意。虽然最终 Ross 用来收集反馈的版本需要比这个复杂一些，但我真的手痒痒了，想现在就做个交互页面出来试试。”

艺术家匹配是一个非常容易的切入点，因为需要的信息仅仅是 FancyVideoService 上的视频标识符、歌曲名和艺术家名。从 Ross 的博客中抓取几十个视频，就可以作为很好的样本数据了。

“那存储方面该怎么办？我是不是应该准备一个……”

你还没说完，Samara 就打断了你的话，因为你忘了 YAGNI 原则。

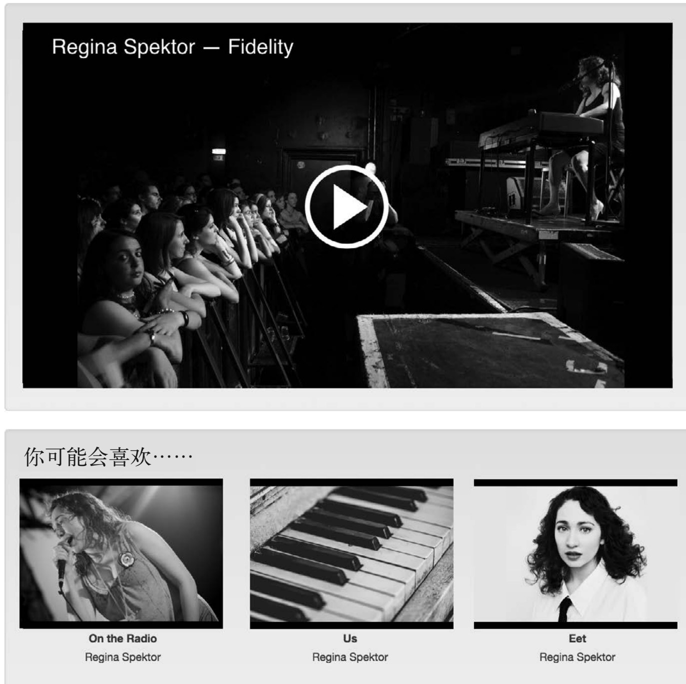
“现在还不需要，我们现在把测试数据集硬编码一下，生成一个队列，然后遍历这个队列就可以得到推荐数据了。”

“这花不了多少时间吧？”

“没关系。做这个工作不用花太多时间！”

Samara 已经准备就绪，所以你把编写代码的工作交给了她。同时，你去整理视频，把一堆歌曲名、艺术家名和视频标识符对应到一起。

15 分钟后，你们俩已经做出了一个像样的半成品，[3] 并把它发布到了网上。



为了展示工作进展情况，你在这一天结束前给 Ross 发了最后一条消息。

> **你**：嘿 Ross，你现在可以看一下那个网站。我们已经加了一些功能，它现在看起来像个音乐推荐系统了。系统现在实现的功能非常有限（只能进行艺术家匹配），但我们还是想给你看看，应该挺好玩的。
>
> **Ross**：哇！做得真不错！交互的感觉的确比呆呆地看着一堆占位图好多了，而且做出来的这个效果几乎和我设想的一样。
>
> 我猜你俩明天会添加一些更有意思的推荐行为吧？不需要做得多精致，但估计会比艺术家匹配复杂点儿吧？
>
> **你**：没错。我们还在考虑具体加什么，但是明天做出东西后会展示给你看。
>
> **Ross**：太好了！再次感谢你们所做的努力。我非常高兴能在工作开始的第一天就看到这个想法的雏形。

注 3：图片来源：钢琴（http://pbpbook.com/piano）；Regina #1（http://pbpbook.com/reg1）；Regina #2（http://pbpbook.com/reg2）；Regina #3（http://pbpbook.com/reg3）。

你和 Samara 做出来的这个“行走的骷髅”[4] 在今后的版本迭代中会变得越来越有趣。

第一天所做的大部分工作是搭建一个基本框架，因此接下来的开发工作就会变得相对容易。你可以在这个已有的框架中发掘真正的问题，并试着去解决。如果一开始就直接考虑整个问题怎么解决，你就会很难找到一个切入点，这样停滞的时间就会更长。

一天的时间已经所剩无几，你决定利用下午时间解决一些小事，上网读读博客，再用手机玩玩小游戏。

1.7 谨记原型并非生产系统

在拖延了半个小时后，Samara 打破了眼前的平静。她带来一个激动人心的消息。

“快看，FancyVideoService 的客服给我们回信啦！”

“是吗？我都不知道你给他们发邮件了。你什么时候发的？”

“你和 Ross 谈话的时候。我想着这种事早联系比晚了好，但没想到这么快就能收到回复。”

你在她身后看着屏幕上的邮件，里面写道：

> 您好 Samara，
>
> 针对我们托管的视频，用热链接访问缩略图，这在技术上不违反我们的规定，因为我们很希望自己托管的视频能够通过各种渠道被大范围分享。
>
> 虽然如此，但我们没有正式支持这种链接方法，所以不能保证视频 URL 的生成方案一直不变。同时，我们有权禁止任何可能滥用服务的行为。至于何种行为属于滥用，我们保留判断的权利。
>
> 更好的方式是在我们的开发者社区注册一个账号，然后使用我们提供的数据 API 作为访问接口。以这样的方式查询缩略图 URL，即使我们更改了 URL 的结构，您的代码仍然能正常工作。
>
> 注册开发者账号的另一个好处是，如果您在不知情的情况下违反了我们的服务条例，我们会马上进行通知，并给您提供帮助来解决这一问题。
>
> 希望以上内容对您有所帮助，祝您“精彩”（Fancy）每一天！
>
> Sarah

注 4：“行走的骷髅”（walking skeleton，http://pbpbook.com/skel）指的是以端对端的方式小规模实现某个特性。它可以作为项目的一个很好的切入点，有助于设计和演化系统的剩余部分。

得知所用的方法可行，你便放心了，虽然你用的并不是 FancyVideoService 提供的标准方案。

为了节约时间，你决定先用不受官方支持的方法来生成缩略图 URL。但是你做了标注，好让这一模块的后续开发人员知道有这么一回事。

今天的进展不错，于是你开心地回家了。

1.8 巧妙设计特性，轻松收集反馈

第二天一早，你刚到办公室，就发现前一天晚上还空空如也的白板上已经贴满了便签条。Samara 在干什么呢？你很好奇，于是开始研究那些便签条的内容。

“哇！ Ross 肯定会爱上这个的！你今天早上几点来的？”

“大概一个小时之前。这个点子是我吃早饭的时候想到的，于是我就决定赶紧过来实现一下看看。”

Samara 看起来有些疲惫，好像没睡够觉，但是你看到她的点子后太开心了，觉得没必要提睡觉的事情。

“那么，我们现在开始做这个吧？我觉得这个想法大有前途，而且你的便签条写得太好了。”

“已经做好了，自己看看网站。”

你坐下来，花了几分钟尝试那些新特性。全部运行良好，尤其对于第一次更新来说已经很不容易了。

“做得这么快，你是怎么做到的？我猜你和往常一样，在实现的时候精简了一些细节。但是即使这样，让我来做的话估计一个小时也做不完呢。”

“哦，你绝对不想看这部分代码。看到这一堆推荐评分了吗？我把这么多东西全存到同一个浏览器 cookie 里了。”

想要做到张弛有度，知道什么时候该快、什么时候该细，是需要经验的。不过你相信 Samara 的判断。你又联系了 Ross，让他查看一下新特性和提供一些反馈。

> **你**：嗨 Ross，很高兴我们又做了新东西给你看。请你抽空去看看那个网站，看完后我再给你讲解一下。
>
> Ross：那我尽快看看，多谢。真没想到你们在午饭前就能给我回复，这真是个惊喜啊！

你：这儿有一张截图。[5] 你看过一些视频后，网站就会是这个样子。请一定亲自感受一下真实的效果。:-)



Ross：刚刚花了几分钟尝试新特性。真是太棒了！

我注意到“感兴趣”那个侧边栏，昨天我们都没讨论过。能否解释一下它有什么作用吗？

你：没问题。先说明一下，我们现在添加这个侧边栏，并不意味着以后它必须成为界面的一部分。

因为推荐系统的工作原理不太好解释，所以我们就做了这个侧边栏，以便形象地向你展示用户在选择视频的过程中，应用是怎么给标签评分的。每个标签都可以点击。如果你点击某个标签，系统就会随机选出对应分类中的一个视频。你可以通过这种做法改变评分，进而改变推荐行为。

Ross：能给我演示一下这个怎么用吗？

你：当然可以。你可以多点几次 Thelonious Monk 那个标签，看看会怎么样。

注 5：图片来源：Ella Fitzgerald（http://pbpbook.com/ella）；Beck（http://pbpbook.com/beck）；Thelonious Monk（http://pbpbook.com/monk）；Regina Spektor（http://pbpbook.com/reg2）。

Ross：啊！我点击完之后，反民谣音乐[6]被推荐得越来越少，而爵士乐变多了。最后，系统给我推荐的视频只剩 Thelonious Monk 的了，我猜这就是你们的用意吧？

你：没错。现在感觉明白点了吗？

Ross：我觉得已经很明晰了，而且还想多试一会儿。另外，我感觉这个系统已经足以示人了。我想今天就把它发给几个人看看，收集一下其他人的反馈。

你们做的这个侧边栏真的非常非常好，因为我如果只听你们的描述，可能很难理解推荐系统的工作原理。真的很感谢。

你：这主意是 Samara 想出来的。我们真应该好好做做这样的功能，因为这有助于理解后台行为，而且你能有机会研究一下我们实现的规则。

* * *

Ross：我再问一个问题就把网站发给别人看：样本数据是从哪里来的？

你：目前我们还只用了很少的数据，是从你的博客里手动挑出来的。但是我们打算好好研究一下选数据这个功能，好让你能自定义数据。

系统现在是从一个 CSV 文件里读取数据的，你可以通过表格软件编辑它。你看一下，这是目前系统里的一些记录。

q97xzziKqOI	Charlie Parker	Chasin' the bird	Jazz	1947
zre0u5XyNfY	Thelonious Monk	Round Midnight	Jazz	1944
oslMFOeFoLI	Dizzy Gillespie	Groovin' High	Jazz	1946
9KwLWpU0_K0	Ella Fitzgerald	How High the Moon	Jazz	1947
tHAhnJbGy9M	Regina Spektor	On the Radio	Antifolk	2006
fczPlmz-Vug	Regina Spektor	Us	Antifolk	2004
MMEpaVL_WsU	Regina Spektor	Eet	Antifolk	2009
4RJob0jSCX4	Regina Spektor	Dance Anthem of the 80s	Antifolk	2009
Z6XiO0o2R7M	Beck	It's All in your Mind	Antifolk	1995

你：第 1 列是每个视频的标识符，出现在 FancyVideoService 视频 URL 的结尾。第 2 列是艺术家名，第 3 列是歌曲名。后面的每一列都可以任意作为标签。现在我们只定义了两个标签（流派和发行年），你可以自己再往后添加标签，想加多少加多少。

注 6：反民谣（antifolk）是一种结合民谣和朋克的音乐形态。——编者注

Ross：等一下……你看我这样理解对不对：如果你把这个表格发给我，我编辑这个表格，随便往里面加视频，然后你可以直接导入我加的视频，连同标签显示在系统里？

你：没错，基本的想法就是这样。刚开始的时候可能要编辑得谨慎一点，因为这些内容的格式一定要正确，但是我们可以适时为你提供帮助。

我们现在的想法是，你再从博客里挑出几百首歌做成一个列表，这样就能对现在的推荐行为进行更实际的测试。

完成这一步之后，我们就可以商讨一下如何把你所有的 4000 多首歌从博客中自动推送到系统里。但是我们觉得这个问题不着急，可以稍后再考虑。

Ross：太好了。我马上就按你说的做，从博客里选一些歌，然后做一个不太长的歌曲列表。弄完之后，我争取今天让几个人看看你们做的系统，下午晚些时候我应该就能把他们的反馈告诉你们。然后我们就能知道下一步应该重点关注什么问题。

我真的不知道该怎么感谢你们。你们做得太好了。

你：做这个项目很有意思，而且你对我们帮助很大，让我们做起来轻松很多，所以也要感谢你。

虽然你们互相表达了感激之情，但接下来并不一定会一帆风顺。俗话说“细节决定成败”，随后的几次迭代会涉及更多细节。在原型阶段结束之前，应该至少会有一个始料未及的重大问题浮出水面。

但出现这样的情况并不意味着开发过程有问题，你恰恰应该做好准备面对反馈环的加速所产生的这一副作用。原型可以帮助你更快地做出有用的产品，但也可能让你失败得更快。如果不用花太多时间就能辨认出死胡同，那你就有更多的时间和精力去寻找正确的路。

不过眼下，你和 Samara 都很满意，打算庆祝一下初步的成功。在项目早期建立起一些诚意和信任，可以给人一种动力，让人在啃到任何创新工作都无法避免的“硬骨头”时，能够坚持下去。

忠告与提醒

- 多向项目参与者提一些能够发掘其目标的问题。这样一来，你既可以验证自己的想法，又可以更好地了解他人对问题的看法。
- 绘制线框图（草图）可以清晰地和他人探讨应用的结构，不会因为被样式细节绊住而停滞不前。
- 一定要在一开始写代码的时候就搭建一个测试系统，让大家都能与其交互。测试系统不需要完善到满足上线要求，只要适合收集有用的反馈即可。
- 在项目早期，集中精力解决有风险或未知的问题。建立原型是为了探索问题空间，而不是为了做出完整的产品。

问题与练习

问题 1：本章中的音乐推荐系统开发工作进展得比较顺利。是否存在可能会出问题（但在本章中没有）的环节，使开发工作变得更为困难?

问题 2：在本章中，开发人员为了顺利开展工作砍去了很多细枝末节，请举出两个例子。在这些决策中，开发人员是如何权衡利弊的？换句话说，开发人员为了换取速度，放弃了什么?

问题 3：假设客户不满足于简单的推荐系统，而是想实现一个涉及机器学习算法的高级系统。那么原型阶段的方向要作何改变?

练习 1：绘制几张线框图，描述一下维基系统的基本功能。然后重复该过程，但这次画一张不同界面的图。思考二者在实现细节方面的不同。

练习 2：随便找一个你最近用过的软件工具或访问过的网站，假设自己现在要把它从零开始实现出来。用不到一小时的时间思考前几步该做什么。

哇，你已经读完了第 1 章！太棒了！

请尽情享受以下这个无关主题的解谜游戏，[7] 将它作为我对你努力的肯定吧。

>	22	#	6F	!	>	AF	#	CA	#	34	>	A9	>	A2	00
00	00	>	A1	00	00	#	34	00	42	42	00	42	FA	F2	FE
?	21	#	68	>	57	42	3D	FA	F2	FE	42	3D	87	00	87
00	00	>	A1	00	00	#	34	00	00	3D	FA	F2	FE	00	87
42	3D	FA	F2	FE	?	5A	00	87	?	17	00	87	FA	F2	FE
00	00	3D	3D	3D	00	42	00	3D	FA	F2	FE	87	#	00	?
42	3D	FA	F2	FE	42	3D	00	00	>	A1	00	00	#	34	00
31	21	00	21	21	#	65	#	6C	>	CC	21	FA	F2	FE	45
?	FA	F2	FE	?	02	00	00	>	A1	00	00	#	34	00	45
31	31	00	FA	F2	FE	?	17	FA	F2	FE	21	21	00	00	45
31	FA	F2	FE	00	00	00	00	>	A1	00	00	#	34	00	87
?	31	00	?	02	00	00	00	>	A1	00	00	#	34	00	FA
31	?	31	FA	00	FE	00	00	?	17	00	FA	#	6C	>	20
31	#	00	00	>	A1	00	00	#	34	00	#	FA	FA	F2	FE
FA	F2	FE	00	FA	00	00	?	17	00	FA	F2	FE	CF	?	FA
FA	F2	FE	00	00	>	A1	00	00	#	34	00	FA	00	?	00

从头开始做，“砰”地一下就能做出来！[8] 必要时在表中进行跳转，但是不要被其中的噪声字符所迷惑。如果你仔细观察，一定能猜出其中的隐藏信息。

注 7：解开这个谜题不需要写代码，但是请准备一张 ASCII 码表，应该会用得着。一旦你掌握了这个游戏的规则，用纸和笔几秒就能找出答案。

注 8：这是双关语，亦表示从左上角的“>”符号开始，遇到“!”终止。——译者注

第2章

观察增量变更，发掘隐藏依赖

假设你所在的公司以其基于高质量文档的大型知识库而闻名。

你所在的公司非常幸运地拥有忠诚的客户。其中有很多客户会撰写博客和文章来分享自己的想法，为充分利用公司的产品献言献策。

为了鼓励这种基于社区分享学习资料的发展模式，你需要搭建一个公开的维基系统，使它与官方知识库网站协同运行。

你更希望将这个功能作为独立的项目来实现，但出于某些无法完全解释清楚的“战略原因”，产品经理希望你把这个新维基系统的特性整合到现有的知识库中。这个维基系统会有自己的“地盘”，但需要和主站共享代码库和基础设施。

该项目的难点在于，如何让维基系统上线，但不对现有的网站产生任何不良影响。表面上看，这很容易，因为加入新特性并不需要改变原有的代码。但实际上该项目暗流涌动，很多深层次的问题有待挖掘。

> **本章主要内容**
>
> 你将了解到，在逐步扩展代码库的过程中，哪些出乎意料的问题会从天而降。

2.1 不存在所谓的“独立特性”

你用数天时间搭建了一个极简的小型维基系统，而且把其中的特性做得和现有知识库系统

很相似。两个系统之间只有一个主要的不同点：只有几个受信任的管理员用户有使用原系统的权限，而任何访问你的公司网站的人都可以编辑新系统的内容。

为了给维基系统收集一些早期反馈，你把它展示给产品经理 Bill 看。Bill 花三分钟尝试了一下，然后转头对你说："太太太棒了！你周五前就把这个系统上线，可以吗？"

这么急着上线新特性似乎很不合适，但你想尽自己所能，迎难而上。于是你坐下来，开始思考该特性上线后可能会出现什么问题。

乍一看，好像并没有什么需要担心的问题。维基系统在网站上有自己的"地盘"，而且你在添加新特性的时候，有意避开了对原有内容管理系统的代码做任何修改。即使维基系统本身土崩瓦解，对网站的其他部分又能有什么影响呢?

过了一会儿，你发现了一个值得关注的问题：不加限制地允许任何人新建和编辑页面，从存储的角度来说有极大的风险。

需要考虑许多可能发生的攻击行为，从创建超大文档以占满所有的存储空间，到创建大量小文档，再到飞速创建文档使存储机制过载。

因为知识库和维基系统使用了同一个存储机制，所以只攻击维基系统就能把原有知识库一起干掉。这就是基础设施层面存在依赖关系的一个例子。这种依赖在你刚刚为原有代码库引入新变更时，不会表现得很明显。

这个想法使你注意到另一个重要的问题：这两套工具是由同一个 Web 服务器托管的。一个效率并不高的同步进程负责把维基系统中的文章由 Markdown 语法转换为 HTML 格式。攻击者根本不需要等到存储机制崩溃，只要不断向 Markdown 转换器发送请求并使其过载，进程便会停止工作。

在这些想法的启发下，你用了几个步骤来降低风险。这几个步骤其实并不怎么复杂，但是可以帮助你避免严重灾难。

- 你把最大页数限定为不超过 1000 个文档。
- 你把每个维基页面的大小限定为不超过 500KB。
- 你把维基系统的 Markdown 进程放到一个队列里，并把队列大小设置为不超过 20 个挂起任务。当队列过载时，会发出"请重试"的出错信息。
- 你加入了监听器，以便监管维基页面的创建、删除和编辑。这些操作发生得太频繁时，系统会发出警告。
- 你为知识库网站加入了有效性监管机制，每分钟进行两次 ping 操作，以保证网站一直能够有效访问,并在可接受的时间范围内响应请求。这项工作本应该在很久之前就做好，但现在由于对改良监管机制有明确的需求，因此这时候做再合适不过了。

这些方法本身并不足以保证系统绝对安全。然而，花上大约一小时，预防一下由共享基础设施引发的最基本的风险，是非常值得的。

对代码做了以上更改后，你很有信心，认为系统现在安全多了。于是你通知了 Bill，表示工具已经可以上线了。

2.2 两特性同屏必相互依赖

几周过去了，你的维基系统运行良好，并未出现任何重大问题。

在最初的版本上线后，你马上又接到新的项目。所以，你在很长一段时间里根本没去想维基系统的事。但就在今天早上，你收到了营销部门的同事 Sandi 发来的一封邮件，你的注意力又得被拉回到那上面去了。邮件内容如下。

> 程序员朋友你好，
>
> 不知道最近你有没有看维基系统的数据分析表，但我们能从中看出，数字开始有所增长。
>
> 在看数据分析表的时候，我注意到一个问题：虽然我们在维基系统里有接近 80 个页面，但大多数人似乎只访问直接从我们最火的广告页面链接过去的文章。
>
> 如果不太麻烦的话，我希望你能花点时间，加一个新特性，帮助客户多了解一下这个网站。
>
> 我希望能加一个侧边栏，列出五个最受欢迎的页面、五个最新的页面、五个最近更新的页面和五个随机选出的页面。
>
> 我们想把这个新特性写进公司的月报里，最近几天就要发布了。所以希望你能挤一挤时间，最好赶在月报发布之前做出来。
>
> Sandi

新加一个侧边栏是合理的需求，而且不是特别麻烦。但和往常一样，你又得赶着做这个工作，这让你很紧张。这些赶出来的工作会在以后出问题时反过头来咬你一口吗?

你也许应该告诉 Sandi，实现这个特性需要更多的时间。而且，任务晚一点完成，对任何人来说都不会引起什么大问题。但在告诉 Sandi 这些想法之前，你决定先快速分析一下，看看自己在当前的情况下能做到什么程度。

添加一个侧边栏不需要对已有的行为进行任何修改，除了要改一改维基网页的 UI。理论上讲，这个变更的风险不大。但实际上，你知道根本没有零风险的事。

仔细看了看 Sandi 的需求，你意识到，列出五个最新的页面、五个最近更新的页面和五个随机选出的页面很容易，因为所有需要的信息都能通过简单的数据库查询轻易获得。找出最受欢迎的页面比较复杂，所以你先把它放在了一边，集中关注更容易实现的需求。

你编写了相关的查询代码，把代码扔到一个有点丑的小侧边栏里，并将它放在了维基页面的右侧。拼凑这些东西只花了大约 20 分钟，但效果出奇地好。你把侧边栏包装进一个“特性内测器”[1] 里编译了一下，这样它就只能被开发人员看到了。两分钟后，新特性上线，可以去试一试了。

第一次访问维基页面时，侧边栏似乎运行得很完美。侧边栏里包含了一系列页面链接，每个链接旁边还有各自的时间戳，用于显示页面的最后修改时间。

但在刷新几次页面之后，你就遇到了第一个问题：页面完全无法加载，取而代之的是显示着“对不起，出错了”的页面。这个似乎独立的变更居然引起了整个维基系统的崩溃！

你查看了一下邮箱，果然，系统已经发来了异常报告。你很快就发现了问题的根源：有几条旧记录的“最后修改”时间戳的键值为 null，而这些值在你开始追踪页面修改时间之前就已经生成了。

直到几分钟之前，这个问题才浮出水面，因为之前这些时间戳从未在 UI 里出现过。要解决这个问题很简单：用控制台将所有值为 null 的时间戳全部设置为维基系统上线的时间，然后加入一个限制命令，以防今后再创建时间戳的值为 null 的记录。

你应该从这次失败中学到的是，对数据库模式所做的任何修改都应该考虑到数据的一致性。不管新特性在代码层面上多么独立，在数据层面上仍有可能存在隐藏的依赖关系。这就意味着，为了支持某一特性而在代码库的某一部分进行的模式升级，可能会使一些看起来毫无关系的其他特性崩溃——这正是你所遇到的情况。

你快速解决了时间戳问题，然后继续点击浏览器的刷新按钮，进行测试。点击了五六下之后，你又遇到了一个严重问题，但这个问题很好解决。

你最初将侧边栏的宽度设计为可调整的，以便当列表中出现比较长的页面标题时，侧边栏可以稍微自动扩展一下宽度。但是这个想法其实非常不成熟，因为你根本没考虑到真正的维基系统中会有一些超级长的页面标题，比如“如何使用专业页面活动工具 WidgetProFlexinator 去做一些你根本想不到的酷酷的事！”

如果允许侧边栏通过自动调整宽度去适应那些极长的标题，则页面内容本身就会被挤作一

注 1：“特性内测”（feature flipping）是一种向特定用户发布新特性的技术。特定用户有可能是某个开发人员、一组测试人员或网站的一部分真实访问者。已经有很多开源库支持这种工作方式，所以应该很容易找到一个兼容你所喜欢的程序语言的库。

团，根本没法看了。这真是蠢得让人想笑，但也值得引以为戒，提醒你别忘记还有一个很微妙的依赖关系：如果两个特性在同一个页面显示，就得做些测试，检查一下它们之间会不会互相影响。

于是，你为侧边栏设置了最大列宽，然后重新进行了部署。然后，你又点击了很多次刷新按钮，直到确定维基系统的所有页面都至少在侧边栏中出现了一次。一切好像都进行得那么顺利。

你更改了一下特性内测器的设置，让 Sandi 也能看到侧边栏。然后，你给她发了一封简单的邮件，把你的进展告诉她。

> 嗨 Sandi，
>
> 关于“最受欢迎的页面”列表，我还要花些时间想一想，但是我们这边已经发布了一个测试用的侧边栏，里面包含你的所有其他要求。这个特性现在只有开发团队和你能看到。请你测试一下，然后告诉我们你的想法。
>
> 你的程序员朋友

在收到 Sandi 的邮件后的一小时之内，你不仅做出了可供她测试和反馈的功能，而且发现并修复了一个数据一致性方面的小 bug。你对自己的工作感到很满意，于是决定休息一下，出去散个步。

2.3 避免不必要的实时数据同步

当你回到办公室时，Sandi 的回复已经发过来了。

> 你好啊程序员朋友！
>
> 在功能上，这个侧边栏和我们需要的已经很接近了。但有两点还是想稍微提醒一下。
>
> (1)“最受欢迎的页面”列表挺重要的，因为现在大家主要通过手动搜索，或点击社交媒体上分享的具体的页面链接进入维基系统。虽然每个页面各自的访问量都不小，但现在它们之间还没有任何联系，所以我们想改进这一点。
>
> (2) 你能把“浅褐色文字加亮绿色背景”改成别的配色方案吗？我觉得最好能把侧边栏做成和知识库主站侧边栏一样的颜色和风格，但稍微改改，只要不闪瞎眼就算进步了。:-P
>
> 有没有可能把这些问题都解决掉，然后周四之前上线？
>
> Sandi

你经常故意把一些尚未完成的特性做得简略一点，免得别人以为它们已经准备好上线了。但 Sandi 说得有道理——亮绿色太过分了。在继续改进之前，你听从 Sandi 的建议，用几分钟改了改代码并发布了一个新版本，把故意弄丑的配色改成了和知识库差不多的风格。

然后，你开始考虑如何为受欢迎度评级。为了实现这一特性，你需要从站点分析服务拉取数据。其实可以实时搜索特定时间段内五个访问量最大的页面，但这会导致每次加载维基页面时都调用一次 API，这非常浪费系统资源。更糟糕的是，这种方法会引入对外部服务的不必要的依赖。

经验告诉你，集成外部服务一般是非常让人头疼的工作，因为会出现各种奇奇怪怪、令人十分不悦的问题。你得提前考虑到，所有的服务集成都可能出现以下问题：响应慢，因评分限制问题而拒绝请求，频繁出现停机故障，返回空响应或格式不正确的响应，触发超时错误，等等。即使上述问题都没出现，迟早也会冒出其他问题，反正就是不让你好过。

如果实在需要操作实时数据，那就没别的办法了，乖乖地花大量时间和精力去写健壮的、容错能力强的代码吧。但在这个案例中，如果每天只把页面访问量更新几次，受欢迎度评分依旧是比较准确、可以接受的。因此，正确的解决方案就是写一个小脚本，在其中设定计划任务。

于是你写了一个脚本，连接分析所用的 API，查询每个页面的统计数据，然后把每个页面的总访问量导入应用的数据库。这个脚本将由 cron 每 4 小时运行一次，而且如果发生错误，或者脚本未在既定时间内完成任务，你会收到通知。最重要的是，偶尔发生的错误并不会产生不良影响，因为这段代码运行在主程序之外。最糟糕的后果不过是受欢迎度评分更新得不及时。

用这种方法，你缩小了问题范围，将它变成了一个简单的数据库查询操作。这使得“最受欢迎的页面”实现起来并不比“最新的页面”和“最近更新的页面”复杂。同时，你也避免了向主站应用添加新的设置信息或库，因为脚本是独立运行的，并且只在数据库层与主站应用共享信息。

做完这些工作需要几个小时，但在下班前，你已经实现了功能完备且可用的特性。Sandi 又看了一下网站，告诉你她觉得很满意。于是，你把访问权限开放给一小部分维基用户进行测试，以确保不会出问题。

确定侧边栏工作正常后，你抽出一些时间整理代码，使之更为正式、整齐，因为这个功能在周四就要正式上线了。这些工作都完成后，你把变更发给了所有人。当新的一周开始时，Sandi 在网站的分析数据中发现了一些可喜的变化。这表明你的代码不负她所望，已经发挥了作用。

2.4 复用旧代码，寻找新问题

你已经有 3 个月没碰维基系统了，而且系统基本上运行良好。然而今天，平静将会在一瞬间被打破。

你到办公室时，发现 Bill 正紧张地一边来回踱步，一边打着电话。你只能听到 Bill 说的话，但很明显，出了很严重的事故。

“不是，我们的维基绝对不是草药公司赞助的啊！我们根本没在上面放过任何广告。”

“不不，super-cheap-pills-for-you.com 不是我们公司名下的域名。”

“不是，我们不是想搞什么恶作剧，更不是想损坏公司名声。你怎么能这么说呢？”

“你是什么时候第一次收到关于这个问题的投诉的？就在今天早上？还好，这算是个好消息。我们马上停掉维基，立即想办法修复。”

Bill 挂掉电话，坐到你旁边。他开始向你解释发生了什么事，但你已经着手解决问题了。

“你一提到草药，我就马上把维基打开来看了，”你说道，“现在看起来有个严重的问题：我们在 Markdown 文件里允许使用了 `<script>` 标签，可能还允许使用了其他乱七八糟的东西。我现在打一个补丁，暂时让整个维基自动跳转到维护页面，等我们可以确定具体情况再说。”

把维基转到维护页面之后，你开始写一个脚本，用于侦测 Markdown 文档中的 HTML 标签。这可以帮你确定有多少页面受到影响，以及如何修复。

分析报告显示，在现有的 150 个维基页面中，有 32 个多多少少使用了一些内嵌的 HTML 语法，但其中只有 12 个使用了 `<script>` 标签。如果这个问题没有这么早被发现，情况可能会变得更糟。

你生成了一个全面的页面链接列表，使之与分析报告对应，并把页面分成了 3 组：“无 HTML”“有 HTML 但无 `<script>` 标签”和“有 HTML 且有 `<script>` 标签”。Bill 负责查看“无 HTML”的页面，你则负责查看其他页面。

所有包含 `<script>` 标签的页面都表现出相同的行为。页面先显示一个模式窗口，提示“请稍等，页面将自动跳转到我们的赞助商网站……”，然后就转到了 super-cheap-pills-for-you.com。这实在很恼人，但至少看起来只是个例，而不是一场大浩劫。

至于所有用了其他 HTML 标签的页面，似乎没有什么问题。大部分 HTML 标签似乎都是普通的内容贡献者使用的，他们对 Markdown 格式还未完全理解，于是继续使用早已熟悉的 HTML 标签。一小部分页面中的 HTML 语句有更具体的用途，比如显示表格，或在页

面中嵌入其他网站的视频。这些嵌入代码提醒了你：`<iframe>` 标签也可能被恶意使用，不过至少现在这种情况还没发生。

Bill 把只包含 Markdown 语法的文档全部审查完，并没有发现明显的恶意使用迹象。到现在为止，维基系统已经拒绝所有访问达半小时了，但你对问题有了更好的理解。

你开始恢复维基的部分功能，以求把对客户的负面影响降到最低。你首先把那 12 个受影响的页面中的 `<script>` 标签移除，然后部署了一段代码，允许对维基页面进行只读访问。Bill 打电话给客户支持团队，向他们报告了你们的进展。到这一刻，紧张的气氛似乎已经有所缓解了。

渡过最大的危机之后，你开始分析该问题的根本原因：让一小部分受信任的管理员使用 Markdown 处理器没有问题，但在充斥着大量自动化程序的互联网环境下，这种机制很不安全。

从根源上讲，这其实也是一个依赖问题。你复用了一个为某种目的而合理设置的工具，但并没有考虑到在稍有不同的情况下，这种设置可能会产生不良影响。你这样做时，只考虑到两种使用环境表面上的相似性，而忽略了二者本质上的不同。这使你被表象迷惑，从而做出了错误的判断。这是一个复用代码的反面例子，值得引以为戒。

再深究一下，会发现有一个更微妙的问题。一开始，你没有明确禁用或限制使用 HTML 标签，这相当于默许了对它们的使用。这个模棱两可的做法让内容贡献者认为，HTML 标签是官方支持的特性，虽然这从你的角度来看明显是系统的一个缺陷。

毫无疑问，必须处理潜在的安全风险。防止匿名访问者向维基页面随意注入 JavaScript 代码，是非常有必要的。但是，你也需要把因修复而带来的负面影响降到最低。

仔细思考这个问题之后，你确定，禁用所有 HTML 标签并不是合适的处理方式。虽然使用 HTML 标签的页面只占维基的一小部分，但最受欢迎的一些页面以很有趣的方式使用了 HTML 标签。如果采取这种简单粗暴的处理方式，这些页面的内容就无法恢复。

你仔细研究了一下 HTML 清理库，最终找到了适合用来解决这个问题的工具。它能清理掉所有 `<script>` 标签，限制 `<iframe>` 标签的使用（只允许一些可信任的域名使用），并处理一些可能引发问题的其他极端情况。

为了评估这一变更对已有文档的影响，你将 Markdown 处理器中每个页面的 HTML 输出代码与清理后的输出代码进行比较。大部分文档经过清理后并未发生改变，只有 5 个页面需要在应用新规则前手动编辑一下。

为了确保以后这样的情况不会神不知鬼不觉地发生，你用余下的整个下午写了一些测试代码，把所有能想到的极端情况都测试了一遍。这让你感到比较满意，但你同时也担心，维

基项目不太可能从此就让你高枕无忧了。这种想法在你的脑海中挥之不去，甚至直到下班时，你还在为此紧张。

忠告与提醒

- 不要因为某个变更没有明显改变现有特性，就认为它会向后兼容或绝对安全。相反，应该对隐藏的依赖关系随时保持警惕，即使进行的是最简单的更新操作。
- 注意除代码库之外的大量共享资源：存储机制、处理能力、数据库、外部服务、库、用户界面，等等。这些工具形成了一张“隐藏依赖网”，会给看起来毫无关联的应用特性带来副作用或引起故障。
- 利用限制和验证的方式，在最大程度上防止局部故障对整个系统造成影响。但还要确保系统拥有良好的监控机制，以保证快速知晓和处理突如其来的系统故障。
- 在复用现有的工具和资源时，要尤其注意使用环境的变化。任何对使用范围、性能标准或隐私安全级别的改变，如果不经过仔细考虑，都可能引起非常危险的问题。

问题与练习

问题 1：隐藏依赖（如共享的资源、服务和基础设施）比代码库中模块与函数间明显的依赖关系更难发现，对此应该怎么做？应该怎样使隐藏依赖变得更明显？

问题 2：本章中的许多例子讲的都是简单的安全缺陷。在现实生活中，维基系统可能会遇到其他形式的攻击，[2] 请尝试说出至少一种攻击方式。你所想到的攻击方式是否在一定程度上利用了隐藏依赖？

练习 1：仔细检查你自己在项目中解决过的 10 ～ 15 个 bug，找出部分或全部由隐藏依赖引起的 bug。创建一个核对清单，帮助自己在以后的代码审查中找出类似的问题。

练习 2：选一个你熟悉的代码库，列出其支持的几个特性。然后，绘制一张隐藏依赖网，展示特性之间的资源共享关系。

注 2：更多背景信息，详见“CWE/SANS 评选出的 25 个最危险的编程错误”（http://pbpbook.com/sans）。

第3章

准确识别痛点，高效集成服务

假设你管理着一份面向程序员的教育型刊物。该刊物拥有少量的付费读者，其收入不足以支持你雇用全职员工。

你和你的朋友 Huan 一起维护着一个定制的 Web 应用，为刊物及其订阅者提供支持。因为预算不多，所以该项目主要由一些靠普通开源工具拼凑起来的代码和几个集成的 Web 服务组合而成。

这几年里，你开始意识到，使用自己控制不了的代码其实成本很高，也有很大风险。由于在设计软件时欠考虑，你已经吃了好几次苦头。这让你在进行外部服务集成时变得更加谨慎。

作为年终总结的一部分，你和 Huan 今天要见一面，回顾一下你们在第三方软件集成方面遇到过的几大痛点。

你们俩互相保证，绝对不能把这次讨论变成“该怪谁”的相互埋怨，因为这样做只会引起争斗，不会带来什么启发。相反，你们会重点讨论如何在未来的工作中解决类似的问题，如果可能，最好是完全避免此类问题。

本章主要内容

你将了解到由第三方系统引发的几类故障，同时认识到，如果不将服务集成考虑清楚，会对决策产生不良影响。

3.1　面对小众需求，切记未雨绸缪

“我们要不先聊聊去括号那件事吧？”Huan 犹豫地问道。你的脸马上红了。

那件事实在令你尴尬，但确实也能让你吃一堑长一智。现在，是时候反省一下了。

“去括号”那件事具体是怎么一回事呢？当时你错误地认为，纯文本电子邮件由第三方通信服务发送时，其内容无论如何也不会发生变化。

你发了几封测试邮件，并且粗略地阅读了第三方服务的文档，当时并没有发现问题。但当再一次发送测试邮件时，你发现邮件中所有的“[]”符号都被删除了。

这只是一个小问题，应该不会对没有特殊符号的邮件产生什么影响。但由于需要发送的邮件中包含一些示例代码，因此任何对文本的更改都有可能出问题。而且“[]”符号在代码里经常出现，所以这就更麻烦了。

于是，你联系了第三方服务的客服人员，问他们是否能够解决此问题。他们的回复总结一下就是：“没错，是有这个问题，但是很难解决，因为这种奇怪的行为根植于我们的基础架构里。”

解决这个问题的一个办法是在前括号和后括号之间插入一个空格，这可能会在某些情况下有用，但并不适用于全部情况。有时，你确实需要准确显示“[]”符号，不然，示例代码会出现很难被察觉的语法错误，直接复制就会运行失败。

由于不能通过邮件发送全部内容，因此你改变了计划，决定在网站上直接发布文章。这时，Huan 加入了你的项目（因为你不可能一边搭建 Web 应用，一边为刊物写文章）。

现在回想起来，这种草率决策本来是可以避免的。

如果你在向付费读者发送邮件之前发现括号的问题，就不会这么着急地寻找解决办法了。

你努力思考，试图想出自己当时能有什么别的办法，但没有结果。你告诉 Huan，自己的脑子里一片空白，然后问她有什么想法。

她表示，如果当时能为邮件做一个冒烟测试就好了：“收集一个超大的样本文章库，用邮件投递服务进行测试，确定这些文章是否都能正确渲染。如果当时用这种方法，应该能及时发现括号的问题，而且测试成本不高。”

她的建议不错，但是你觉得有点不自在。你问自己，为什么当时没能想到类似的方法？你的脑子开始转动，你意识到，让你陷入窘境的根本原因是自己的思维过程有缺陷。

> **你**：冒烟测试肯定是有帮助的，但该想到的时候，我总想不到。这才是真正的问题。

理论上讲，在使用任何第三方系统时，我们都不能完全信任它，除非已经证明它确实可靠。但现实是，时间紧迫、预算紧张；我们经常得急着往前赶。

Huan：你的意思是，之所以会遇到问题，是因为你没有仔细考虑就匆忙决定使用某种方法？

你：很明显，我当时是想找一条捷径。我觉得那个第三方服务的名声不错，就选择了它。这听起来似乎合理，但其实不应该那样做。

Huan：但是如果那个服务很受欢迎，应该能说明它不错吧？我感觉如果换成是我，也很容易犯同样的错误。

你：你知道街口那家超赞的汉堡店吗？就是大家都夸，说是世界上最好的汉堡店的那家？

Huan：妙趣汉堡屋！我超爱妙趣汉堡屋！你怎么突然提到它了？

你：你觉得他们家的鱼肉三明治怎么样？

Huan：不知道，我从来没点过。说实话，我根本不知道他们家还卖这个。大家都是去吃汉堡的。

因为 Huan 已经习惯了你这种爱讲谜语的套路，所以她很容易就从中解读出了你的要点：在选择邮件投递服务时，你的表现相当于去妙趣汉堡屋点了鱼肉三明治，而不是汉堡。

现在问题已经很清晰了，你们又用了几分钟深入探讨当时怎样做会更好。

- 应该多做一些调查，看看有没有其他人也在以你的这种方式使用该服务。也许很难找到这样的人，这本身就该给你敲响警钟，让你停下来思考一下为什么没有人这样用。
- 对于这种特殊用法，不应该想当然地认为一定会顺利，这是很危险的。反之，最好是采取和对待其他未知问题一样的态度，保持警惕。注意到风险后，应该进行更为全面的测试，或者应该考虑至少用几周时间进行内测，而不是急着开放注册功能。
- 另一个被忽略的关键问题是："如果第三方服务不符合我们的预期，该怎么办？"这个问题在任何涉及软件系统中的重要依赖时都应该被问到。无论你是想进行纯粹的思维实验，还是想着手制订详实的备选计划，注意这个问题都会对你大有裨益。如果服务真的出现问题，你就不会大惊失色，也不会手足无措。

3.2 谨记外部服务并不可靠

分析第二个案例时，你提到有一次网站的登录系统突然停止工作了，这令你和 Huan 都很吃惊。

这个事件与刚刚分析过的电子邮件问题有一些奇怪的联系。你原本计划把文章直接发送到订阅者的邮箱，这样他们不用再进行任何额外操作，直接就可以开始阅读了。因为你改变了这个计划，所以事情变得复杂了。

受括号问题的影响，你把文章放到了一个网站上。由于你只想把内容分享给会员，因此网站需要引入身份验证机制。

强迫订阅者记住自己的用户名和密码是很差的用户体验，所以你决定采用大部分订阅者已经每天都在使用的一个身份验证服务。这样，你就可以为订阅者分享链接，而他们也不需要再记住另一套登录凭证。

实现这个特性很容易，而且只需要进行一次性开发，然后就根本不用再考虑它了。但好景不长，在你发送刊物的第 35 期时，警告邮件像雪片般从网站的异常报告系统飞来。

在收到第一封警告邮件之后的一小时内，你就找出了解决方案。但不出所料，你的方案并不完善。然后，你找到了一个更为完善的方案，但补丁引入了一个小小的错误，而几天后你才发现这个错误。

产生这个故障的技术原因没有什么特别之处，但你认为，探究一下问题究竟为什么会发生，可以帮助你和 Huan 获得一些有用的见解。为了揭示问题的答案，你建议玩“五个为什么”[1] 的游戏。

Huan 同意当询问人，于是开始向你发问。

为什么身份验证系统会突然无法使用？

应用的依赖库使用了一个旧版本的身份验证 API，而这个 API 最后被停用了。它一被停用，我们的登录功能马上就无法使用了。

为什么你要用过期的第三方服务？

身份验证是我们的 Web 应用实现的第一个特性，当时运行得很好，没出现任何问题。从那以后，这部分代码在日常开发中就看不到了。

没有人想到 API 会被停用，更没有人想到它会悄无声息地被停用。

为什么你想当然地认为 API 永远不会被停用？

集成之后，第三方服务一直很正常。没人考虑其中的实现细节或它的服务条例。

注 1：“五个为什么”（http://pbpbook.com/5whys）一般用于发掘问题的根本原因。做法是通过反复询问“为什么”来揭示问题的大背景。由于大部分问题的根本原因不止一个，因此为了从不同角度发掘原因，可以根据需要，不断重复询问的过程。

开发这个特性时，我们没有考虑清楚集成第三方库与集成第三方 Web 服务有什么不同。

过期的第三方库仍然可用（只要代码库及其基础设施中没有引入不兼容的变更）。由于预算紧张，因此我们的维护策略是，只在非常有必要的时候（比如打安全补丁或出现其他大问题时）才对库进行升级。

然而，Web 服务是一种完全不同的依赖。因为服务必然涉及与远程系统的交互，而这个远程系统又完全不受我们的控制，所以服务随时可能被变更或停用。在项目的维护计划里，我们没有把这点考虑进去。

其实这个问题也不是完全没人想到，但是我们以为，既然这个提供身份验证 API 的服务这么火，如果提供方决定对其进行重大变更，或者停掉它，一定会通知用户。

为什么你以为 API 提供方会通知用户？

现在回想起来，很明显，每个公司都有自己的规定。除非文档明确解释了服务变更的通知方式，否则我们没有理由假定会收到关于重大变更的通知邮件。

说到这一点，我还想到一个造成混乱的原因。应用中使用的库并不是由服务提供方来维护的，而是由第三方创建的。在我们集成它之前，已经有一个 API 版本了。从服务提供方的角度看，这个工具只是一个旧版客户端。

维护身份验证 API 的那家公司宣布变更的方式也不正式，只发了几篇博客文章，而这些文章藏在数百篇与之毫不相干的文章里，根本不好找。他们还有一个 Twitter 账号，但这个账号是在 API 停用很久以后才创建的。如果一开始就研究好如何接收此类通知，我们就能在服务变更对客户产生负面影响之前有所行动。

为什么你没有在刚实现身份验证特性时就研究好如何接收服务变更通知？

这个特性是在邮件投递服务出问题后马上添加的。我们匆忙实现了这个特性，将其作为向订阅者发送文章的替代方法，因为怕延误了刊物的发行计划。

需要做的决策太多了。在这么大的压力下，做什么都没法仔细考虑。如果当时情况允许，我们应该能从邮件投递服务的问题中总结出一些经验：第三方系统本来就不可靠，再受欢迎的系统也是如此。

但在当时的情形下，我们有点盲目乐观，以为第三方服务应该不会出什么问题。至于为什么会有这种想法，只能这样解释：我们在涉及服务的故障方面太缺乏实战经验了。

询问完之后，Huan 总结道，最好审查一下你们正在进行的所有项目，看看它们依赖哪些服务，然后确定如何得知服务发生变更。你们俩达成一致意见，决定先抽出一些时间做这件事，然后再继续讨论。

3.3 服务一旦有变，查找过期的模拟对象

尽管“五个为什么”这个练习确实挖掘出了几个有趣的点，但并没有完整地揭示到底哪里出错了，以及为什么会出错。继续对话题展开讨论后，你又一次意识到，测试策略的缺陷是导致问题出现的一个原因。

找到身份验证失败的直接原因和解决方案很容易，因为很多人和你一样，也在依赖被停用的 API。在网上搜索了一番之后，你了解到，需要对 API 客户端进行升级——这个解决方案看起来很简单。

你知道，不做全面的测试就升级一个很重要的库是非常危险的。于是，你抽查了应用的验收测试情况，发现对身份验证的测试还是比较全面的——验证成功和验证失败都测试过。

这些测试结果让你有了信心，而且你发现库在升级之后仍然通过了测试。于是，你认为这次走运了，因为升级工作不需要修改任何代码。手动对开发环境和生产环境都进行测试后，你信心十足，以为一切进展顺利。

然而几天后，与身份验证服务相关的一份异常报告证明你是错的。很明显，你的测试忽略了某个重要因素（至少这是你在数月之后回想时想到的出错原因）。

* * *

你仔细阅读项目的提交日志，以确认自己的想法。果然，你发现在最后一轮故障之后，有一次针对模拟对象的更新。你把这个发现告诉了 Huan，但她看起来并不感到惊讶。

“就知道会这样，”她说，“我们本来应该针对真实的 API 做一些现场测试。因为没有做到这一点，所以我们的测试永远不会像我们想的那么全面。如果在发送每一封邮件前都运行一遍测试，那么我们在客户受影响之前就能查出问题。”

直觉告诉你，Huan 应该说得没错。但是，事后诸葛亮谁都会当，测试的成本可不低。

即便如此，如果你在抽查测试覆盖情况的时候再深入一点，肯定会有好处。模拟对象是在底层搭建的，因此有没有模拟，对验收测试结果的影响都微乎其微。这一点很容易让人忘掉模拟对象的存在。

在升级了库而且没有发现任何故障后，你又手动进行了测试，确认事情如你所想，于是你就想当然地认为一切正常了。自动化测试能够捕获到库接口中的所有意外变更，手动测试则只能验证服务是正常的。

在订阅者登录的常规情形下，手动测试是可行的，也就是说，你在库升级后所做的第一次修复确实解决了现有客户的问题。但要想发现这次修复并不全面，还得过上好几天；当新客户试图注册时，身份验证系统发生了错误。

这个问题非常微妙，不易察觉。对现有的订阅者进行身份验证时，系统只需要一个和数据库中用户记录对应的唯一标识符。而创建新账号时，还需要访问由服务的响应数据提供的邮件地址。

升级之后，数据模式发生了改变，但仅限于与用户有关的元数据；标识符还是保持原状。因此，活跃的订阅者还是和往常一样可以正常登录。然而，如果不升级代码以适应新的数据模式，就无法完成新用户注册。

想了一会儿后，你只得承认 Huan 是对的：现场测试非常重要。

> **你**：你说得很对。如果当时测试了真实的身份验证服务，应该会有帮助。但同时我也希望自己当时能更仔细地抽查测试覆盖情况。
>
> Huan：你觉得当时应该怎么做才更好？
>
> **你**：我不应该只看测试，还应该看看相应的支持代码。
>
> 这样的话，我就能注意到身份验证服务中模拟响应数据的配置文件了。如果当时我看到了那个文件，可能就会去考虑应不应该在升级时把它也修改一下。
>
> Huan：没错，这想法也很好。以后如果再遇到服务依赖变更，我们除了要进行代码复查，查找过期的模拟对象，一定还要定期运行小规模的自动化测试来审查服务本身。
>
> **你**：这计划不错！

3.4 遭遇烂代码，维护必头疼

针对今天的最后一个议题，Huan 建议讨论一件有点不那么切题的事：Web 爬虫在数分钟之内触发了几百个邮件警告。

思考这个问题时，你开始明白她为什么提起这件事：在开放的互联网环境中，你需要担心的不只是自己集成的服务。有时，会有人不请自来地把你和他们自己集成到一起。

你快速查看了一下很久之前的邮件、ticket 凭证和提交记录，以便在头脑中重新梳理此问题的时间线。

- 在一个周三的凌晨 3 点 41 分，你的邮箱收到第一波异常报告。你当时正好没睡，于是马上把爬虫的 IP 地址加入了黑名单。看起来事情解决了，于是你给 Huan 发了封邮件，让她调查一下，然后就去睡觉了。
- 当你在第二天早上醒来时，Huan 已经发现了问题的源头，然后用两行代码打了个补丁，问题似乎已经得到解决。由于一个查询方法实现有误，服务器没有正常报告自己的 404

状态，而是抛出了异常。在正常使用应用的情况下是不会出现这种问题的；一定是某个写得很烂的爬虫发出了奇怪的请求，在服务器上兴风作浪。

- 周五凌晨 4 点 16 分，你又收到了一波相似的异常报告。错误的具体细节有所变化，IP 地址也变了，但那奇怪的请求和你在周三看到的一模一样。虽然爬虫仅在两分钟之内就发出了几百个请求，但两分钟一过，请求就停止了。这没有什么值得高兴的，但事情没有变得更严重，已是万幸。
- 接近周五中午时，你联系 Huan，想知道事情的进展情况，但她没有回复。此时，你已经意识到，目前的问题可能是因为 Huan 在解决之前那个问题之后做了一个小改动（而你在收到第二波异常报告之前根本没有复查她的代码）。
- 周日早上 6 点 24 分，爬虫第三次光顾，并且触发了一波和周五那天一样的异常报告。这次，你自己检查了问题，做了小小的修复。你还直接根据爬虫发出的各种请求，添加了一个回归测试，以确保修复后的代码一直有效。
- 周一，你和 Huan 互发了几封邮件，梳理了一下错在哪里、为什么会错，以及该错误本身会有多么危险。

回忆这一系列事件让人非常不舒服。同一个错误一而再、再而三地犯，脸皮再厚也会感到尴尬。虽然事情已经过去了好几个月，但现在回想起来，还是觉得不安。

> **你**：我很抱歉，自己对这件事处理不当。
>
> 在那些邮件里，我像是在怪你把事情搞砸了。其实很明显，应该怪我沟通不当。
>
> **Huan**：也许这是其中一个原因，但实际上我也应该更仔细些。
>
> 我完全没考虑这个问题会对邮件投递服务产生多大危害，只是想着你因为警告邮件太多而很烦。

你刚开始只关注了表面问题：邮箱被一大堆没用的警告邮件轰炸，这很烦人。这些错误甚至都不是真正的故障，根本不需要关注；它们只是计算机程序运行的结果罢了。

深究一下，你会发现，根本问题是应用的一个端点暴露了，这会触发无数个异常报告。更糟糕的是，你使用的邮件投递服务只允许你每个月发送有限数量的邮件，而这一波波警告邮件是在浪费你的发送次数。

随着调查的深入，你也注意到，异常报告系统使用的邮件投递服务和客户通知系统所用的一样。这样的设计大有问题，它会大大增加上述问题对客户产生不良影响的可能性。

如果不做处理，过不了几周，有限的邮件发送次数就会被这些洪水般涌来的邮件用尽，你也就无法向订阅者发送邮件了。然而，你也许根本不用等那么长时间，因为系统发送邮件的速度与爬虫发送请求的速度一样快。服务提供方完全有可能限制或拒绝你的发送请求，因为你的发送频率不断超过他们的上限。

这个问题其实应该归结于你们的经验不足：你们俩都见过爬虫做一些奇怪的事，但从来没见过它们对服务产生负面影响。

从邮件记录中可以看出，其实你在某种程度上意识到了潜在问题，但没有明确地和 Huan 交流此事。这一点在当时不太明显；只有在今天进行回顾的时候，一切才清晰起来。这次惨痛的教训给你们好好上了一课：在处理紧急事件时，交流顺畅能够决定成败。

3.5 不存在纯粹的内部问题

你用很短的时间温习了一下几个月前写在邮件里的回顾笔记。里面有很多有用的想法，你和 Huan 现在对它们已经烂熟于心了。

- 为发现的所有问题都添加回归测试，无论问题有多小。这一点很重要。
- 务必检查测试配置中的模拟对象，以免测试结果让你错误地认为模拟的 API 客户端仍能正常工作。
- 让异常报告系统和客户通知系统共享一套邮件投递机制，是有风险的。
- 异常报告系统最好能把相似问题合成一个报告,而不是针对每个问题都发一条警告信息。

这些主意都非常好。你们现在已经能清晰地看到问题，并且已经在最近的项目中采纳了上述建议，这是进步的标志。但为什么没有在一开始就考虑到这些问题呢？为什么一定要等到失败了、摔疼了，才去关注它们呢？

> **你**：我觉得所有问题背后都有一个共同原因：我们在维护过程中，本意是好的，但往往被误导了。
>
> Huan：为什么会这样呢？
>
> **你**：这个嘛，我觉得我们精神可嘉。而且我们都同意，只要是客户遇到的问题，无论多小，都应该作为首要问题去解决。但是我们在这上面投入的精力过多，导致对一些内部的质量问题和稳定性问题考虑不足。
>
> Huan：那样做有什么不对吗？我们不应该一直把客户视为上帝吗？我还以为你觉得这是我们的优势呢。

你愣了一会儿，然后组织好语言，道出了你在整个回顾过程中一直在反复思考的问题：

“也许根本不存在纯粹的内部问题。只要我们的代码面向外界、与外界交互，当它出现问题时，就有可能对客户产生影响。如果我们多关注一下系统与外界的交互问题，重视每一个小问题，可能就会有更好的结果。”

整个房间在这一小时的谈话中首次陷入沉默，似乎连灰尘都静止了片刻。沉默过后，你和 Huan 朝着妙趣汉堡屋走去，要尝尝他们家那绝妙的鱼肉三明治。

忠告与提醒

- 当你的应用依赖于你不甚了解的外部服务时，一定要加倍小心。如果找不到和你的用法相似的成功例子，就要提高警惕：往好的方面说，可能没有人用过该服务解决与你的需求类似的问题；往坏的方面说，该服务可能根本就不适合你。
- 谨记库和服务之间最关键的不同点：只有在代码库或基础设施发生改变时，库才会引起显著变更；而外部服务随时都可能让系统行为发生改变，甚至导致系统崩溃。
- 只要服务依赖发生变更，就要在测试中密切关注是否有模拟对象过期。为了防止被测试结果误导，一定要确保针对你所依赖的真实服务运行一些测试。
- 认真对待每次代码复查，把它当作对所依赖服务的一次小规模审查，比如评估测试策略、考虑如何处理故障，或者思考如何防止错用资源。

问题与练习

问题 1：本章中的开发人员被 Web 爬虫的不良行为折腾得很惨。如果有不速之客未经你同意就与你的系统集成，还可能会引发什么别的问题？

问题 2：假设在你的核心业务中有一项服务集成至关重要：没有它，一切都完了。这种依赖会如何影响你的计划、测试策略和维护策略？

练习 1：通读 Richard Cook 的文章“复杂系统是如何崩溃的”（http://pbpbook.com/cooke）。从文章中找出至少 3 个和本章故事相关的地方，再找出另外 3 处，与你自己在工作中遇到的问题一一对应。

练习 2：审查你自己的代码库，找出其依赖的所有外部服务，然后思考当这些服务发生故障时，系统可能会出现什么问题。最后，记下你能想到的所有问题，以便在日后的工作中降低风险，提高软件的容错性。

第 4 章

设计严密方案，逐步解决问题

假设在最近几个月里，你正在指导一位刚刚进入软件开发行业的朋友。

你的朋友 Emma 大概一年前找到了她的第一份软件开发工作。在那之前，她对软件开发的了解主要靠自学。为了能够尽快获得经验，Emma 在偶尔遇到困难时，会向你寻求帮助。

在过去的几周中，Emma 注意到自己好像在遇到定义明确的任务时做得不错，而在遇到包含很多细节、需要自己一一理清再解决的问题时，就有点不知所措了。

发现这一绊脚石后，Emma 问你是否需要做些编程题来训练这方面的能力。

你认真思考了一下这个想法。编程题一般画蛇添足地将实现细节复杂化，故意把数据表示得很麻烦，而且在完全理清其规则之前，是很难解出来的。这就意味着，靠做题很难训练编程实战能力，但编程题很适合用于探索通用的问题解决技巧。

于是，你找了一道 Emma 可能会喜欢的题。她随即开始解题，你则在她身边待命，帮助她解决在解题过程中遇到的问题。

本章主要内容

你将学到几个非常简单直接的策略，帮助你有条不紊地逐步分解并解决难题。

4.1 收集事实，清晰描述

你们今天要研究的问题是“数扑克牌”。[1] Emma 首先开始阅读题干。她用 5 分钟就读完了，这让你有点吃惊。随后，你让她说一下自己的想法。

> **Emma**：我已经了解这道题的基本思想，但还不确定该从哪里入手。
>
> **你**：没关系。先解释一下你现在想到哪一步了，我们可以看看能做到哪一步。
>
> **Emma**：题目描述了某个棋牌游戏的记录，其中列出了每个玩家每一步的出牌方式。需要根据游戏进行过程中的出牌情况，判断每轮结束后特定玩家手里的牌。
>
> **你**：没错，我也是这么理解的。你觉得解决这个问题的难点在哪里？
>
> **Emma**：题目没有给出扑克牌的完整信息，不知道游戏中具体有哪些牌。所以，我猜是不是要用排除法之类的方法，来确定每个玩家手里都有哪些牌？我从这里开始脑子就一片空白了。
>
> **你**：我想了想，感觉这个问题可能太复杂了，单纯盯着题看是想不出什么来的。不如再读一遍题，同时记下一些关键点，然后看看会怎么样，如何？
>
> **Emma**：如果你觉得那样做有帮助，当然可以。咱们开始吧。

排除噪声信息，找到问题的核心，是在遇到任何复杂问题时都应该首先做的工作。你不想唠唠叨叨地说教，于是打算用举例的方式给 Emma 演示这种方法。

你大声朗读题干，Emma 则在一旁记笔记。然后，你们交换角色，重复同一过程。在复查和合并笔记内容之后，这个扑克牌游戏的基本细节逐渐显现出来。

- 此游戏使用一副标准扑克牌。
- 玩家在出牌时可以做以下 4 种动作之一：抽取一张牌；将一张牌传给另一个玩家；从另一个玩家手中接过一张牌；打出一张牌。
- 对于每一个玩家而言，抽牌数、传牌数、接牌数和出牌数没有限制。
- 当一张牌被打出后，这张牌就退出了游戏。

游戏记录列出了每个玩家每次出牌的行为，但针对每个玩家所提供的信息量不同。

注 1：详见 Eric Gjertsen 的“数扑克牌”（http://pbpbook.com/cards）。本章并不要求先读完此题的题干，但如果你希望增强代入感，可以现在去读一读。

Rocky	关于所有牌和出牌行为的完整信息
Lil	列出每一轮的几个可能的动作序列；你需要找出其中正确的序列
Shady + Danny	仅有公开可见的信息，即你知道他们抽牌、传牌、接牌， 但看不到具体是哪张牌。当然，所有人都能看到出牌信息

这道题需要解题者列出每一轮结束时 Lil 手中的牌。为了生成这一列表，需要穷举每次轮到她出牌时她的可能动作，然后找出正确的那个。这就是 Emma 之前提到的“排除法”。现在需要解决的问题是如何实现这一方法。

Emma 用几分钟时间仔细看了看自己的笔记，然后思考这个问题。突然，她灵光一现，发现了一个不错的切入点。

> Emma：啊，我明白了！我们遍历一下每种已知的出牌方式中的动作，然后去掉那些会产生不可能的结果的序列。比如有人抽到一张已经被打出的牌，或者传一张已知在别人手中的牌。
>
> **你**：没错。不过需要提醒的是，对于这种比较难的题目来说，排除法应该不会这么简单。但是你的基本想法是对的。
>
> Emma：我觉得自己需要再研究一下这个问题，然后才能明白究竟复杂在哪里。但是我仔细看了看这道题的一些练习数据集，感觉现在已经明白了。不如我们先这样开始做，看看用现在已知的条件能做到什么程度，如何？
>
> **你**：当然可以！听起来不错。

在此之前，题干只是一大堆毫无意义的细节，输入文件格式混乱，让人一知半解。而现在，Emma 的头脑中已经有一个清晰的解题目标，也对达成这一目标的方法有了初步的思路。她已经能从全新的角度看待此题了。

4.2 写代码之前手动解决部分问题

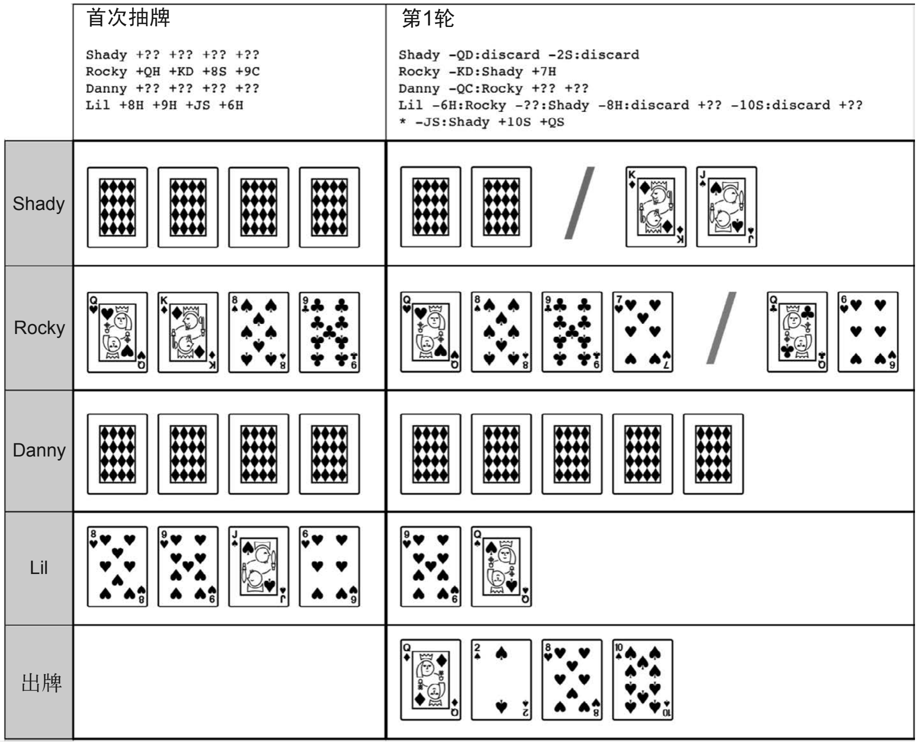
在 Emma 研究第一个练习数据集时，你保持安静。这个数据集主要介绍游戏记录的一些语法和基本结构，所以只包含 Lil 每轮出牌动作中的单一有效序列。

```
Shady +?? +?? +?? +??
Rocky +QH +KD +8S +9C
Danny +?? +?? +?? +??
Lil +8H +9H +JS +6H
Shady -QD:出牌 -2S:出牌
Rocky -KD:Shady +7H
Danny -QC:Rocky +?? +??
Lil -6H:Rocky -??:Shady -8H:出牌 +?? -10S:出牌 +??
* -JS:Shady +10S +QS
Shady +KD:Rocky +??:Lil -KD:出牌 -??:Lil
Rocky +QC:Danny +6H:Lil -9C:Danny -6H:出牌 -7H:出牌 +3D +3H
Danny +9C:Rocky -AD:出牌 +??
Lil +??:Shady +?? -??:Danny -??:Shady +??
* +AH:Shady +8D -8D:Danny -QS:Shady +8C
Shady +??:Lil -7S:出牌 +?? -10H:出牌
Rocky -QH:Lil +5D -8S:Shady -3H:出牌 -QC:出牌
Danny +??:Lil +?? +?? -??:Lil -3S:Rocky -??:Shady
Lil +QH:Rocky +??:Danny -AH:Rocky -QH:出牌
* +4D:Danny
```

10 分钟之后，Emma 打破沉默，向你发出求助信号。

> Emma：我不知道怎么处理这个文件。
>
> **你**：你的意思是你不知道怎么解读这些动作，还是不知道怎么把这些动作用代码表示出来？
>
> Emma：不知道怎么写代码。比如说，我知道 `+QH` 的意思是“抽到一张红桃 Q”，也知道 `-??:Shady` 代表“把一张（未知的）牌传给 Shady”，但我不太确定怎么根据这些数据建模。
>
> 而且，Lil 可能的出牌序列的格式也让我有点不解。我知道那些带有星号的行是为了填充 Lil 出牌时的 ?? 部分，但我还不清楚怎样通过代码来把它们合并起来。
>
> **你**：说实话，我现在也还不知道怎么根据这些数据建模。我通常会在思考如何写代码之前稍微手动推敲一下整个问题，这样做可以帮助我看到问题的关键，以免过早陷入实现细节。
>
> 我们现在就来逐步研究游戏记录，看看能得到什么新信息。

Emma 一条一条地阅读着游戏记录，你则手动绘制了一张表，表中列出了每个玩家手中牌的状态。经过首次抽牌和第一轮完整的游戏后，你得到了如下这张表。



仔细查看这张表之后，此题背后的逻辑变得更清晰了。

- 一手牌指的是每个玩家得到的牌的集合，以插入顺序排列。
- 一手牌中可能会包含还未翻开的牌。
- 传牌动作包含两个步骤。未轮到接牌方出牌时，该动作不算完成。
- 已出牌堆指的是已翻开的牌的集合。牌不可放回，一旦被翻开，该牌就相当于退出了游戏。

用这些基本思想作为引导，Emma 开始着手写一些代码，对“一手牌”和“已出牌堆”这两个概念进行建模。她从输入文件中不同的动作入手，为这些动作写了相应的函数，如 `hand.add("QC")` 表示 `+QC`。

在为这些模型写代码的过程中，Emma 发现，若要更新某个玩家手中的牌，不只是增加或减少集合中的元素那么简单。比如，出牌函数 `discard()` 的行为取决于牌是否被翻开。所以问题就来了：如何对未翻开的牌进行建模呢？

Emma 还遇到好几个与之类似的小困难，她开始有些沮丧了。但这时你提醒她，手动推敲只是为了对此问题的大框架有一个大致的了解。于是她意识到，在具体实现代码细节时，遇到一些很难解决的问题实属正常。

Emma 逐步找出了另外几个手动推敲时不容易发现的极端情况。终于，她的代码在独立测试时正常了，于是她把第一个样本数据集中的游戏记录手动转换为自己的模型中的函数调用。

Emma 运行了自己的脚本，然后惊喜地发现，在第一次尝试时自己的程序就可以输出正确结果。这个小小的胜利很重要，因为它可以给 Emma 动力，让她积极解决剩下的问题。

4.3 核实输入数据，随后进行处理

Emma 把注意力转向第 2 个练习数据集。

```
Lil +5C +2H +8H +6D
Shady +QH +AC +7C +2D +8C +3S -??:Lil
Rocky +KS
Danny +4H
出牌 +4D +7D +JS +6S +6H +2C +5D +3C
Lil +??:Shady -6D:出牌 -??:Danny +?? +??
* +8H:Shady -2H:Danny +JD +2D
* +8C:Shady -8C:Danny +JD +4S
* +QH:Shady -2H:Danny +7D +AS
* +AC:Shady -8H:Rocky +AS +8D
* +8C:Shady -2H:Danny +10H +9H +4C
* -8H:Danny +8C:Shady +4S +AS
```

Emma 注意到，这个文件和刚才那个在格式和布局上略有不同。于是，她重新看了看问题描述，并阅读自己当时的笔记，以确定这个样本数据集的用途。

> 这个文件描述了 Lil 出牌之前的一轮游戏，并给出了每个玩家手中现有的牌和已出牌堆里的牌。现在有 6 个可能的状态分支，其中只有一个是 Lil 的真实动作。确定哪个动作是正确的，并推断该轮游戏结束时 Lil 手中有哪些牌。
>
> 注意，本数据集包含主问题中很多个（但不是全部）无效分支。

Emma 告诉你，她打算从遍历游戏记录入手，列出所有未出的牌，这和你处理第一个数据集的做法类似。

这个想法不错，但还有一点值得重视。你让 Emma 复查一下分支的语法，以确认她在按照自己的计划做之前完全明白其工作方式。

为了验证自己的假设，Emma 创建了一张表，把每个分支中的出牌动作说明完全展开。理论上讲，需要做的无非就是将带有 `??` 的动作与每个分支中的动作对应起来。但实际上，并不是所有的分支都使用了正确的格式，如下所示。

Lil +??:Shady -6D: 出牌 -??:Danny +?? +??		
+8H:Shady -2H:Danny +JD +2D	→	Lil +8H:Shady -6D: 出牌 -2H:Danny +JD +2D
+8C:Shady -8C:Danny +JD +4S	→	Lil +8C:Shady -6D: 出牌 -8C:Danny +JD +4S
+QH:Shady -2H:Danny +7D +AS	→	Lil +8H:Shady -6D: 出牌 -2H:Danny +7D + AS
+AC:Shady -8H:Rocky +AS +8D	→	格式错误！（传给了错误的玩家）
+8C:Shady -2H:Danny +10H +9H +4C	→	格式错误！（抽了3张牌）
-8H:Danny +8C:Shady +4S +AS	→	格式错误！（动作顺序有误）

Emma 对这一发现感到很吃惊。你们俩稍微聊了聊这个问题，因为其中有一个很重要的地方值得引以为戒。

> Emma：这好像是故意给我们添麻烦，根本没有为解题增添什么乐趣。为什么出题的人要在这一点上耍我们呢？
>
> **你**：我想为出题者辩解一下，因为这样做使得这道题更为真实。现实中的原始数据一般都是一团乱麻，所以在使用样本数据前对它进行处理，这再自然不过了。
>
> Emma：你是说你早就知道样本数据会有问题，所以让我手动确认一下？
>
> **你**：不是，这只是为了让你养成好习惯，否则很容易写出“垃圾进、垃圾出”的程序。
>
> 只要你找到一个格式不正确的分支，就能知道需要检查所有分支的格式。发现这一问题只需要几分钟，但磨刀不误砍柴工，这时间花得很值。
>
> 当然，在某些情况下假设数据格式正确是没问题的，但如果你不能确定，最好还是谨慎一点。
>
> Emma：好的，现在我明白了。谢谢。

为了编写检查格式的函数，Emma 需要处理 Lil 的出牌数据，并提取出所有带 ?? 的动作。你建议把这组数据转换成一组数组，以便在代码中更好地表示其结构。

```
Lil +??:Shady -6D:出牌 -??:Danny +?? +??

[[:接牌, "Shady"], [:传牌, "Danny"], [:抽牌], [:抽牌]]
```

所有的分支都可以用相同的方式进行转换。这样处理后，如果能够生成相同的结构，那就说明分支的格式至少是正确的。如果结构不同，该分支则可以立即被标记为无效。

Emma 查看了样本文件中的 6 个可能的分支，并手动将文字转换为你所建议的格式。做这些工作时，她发现前 3 条数据符合 Lil 的出牌动作结构，而后 3 条都不符合。

结构正确（+??:Shady -??:Danny +?? +??）

```
[[:接牌，"Shady"]，[:传牌，"Danny"]，[:抽牌]，[:抽牌]]
```

传给了错误的玩家（+AC:Shady -8H:Rocky +AS +8D）

```
[[:接牌，"Shady"]，[:传牌，"Rocky"]，[:抽牌]，[:抽牌]]
                  ^^^^^^^^^^^^^^^^
```

多抽了牌（+8C:Shady -2H:Danny +10H +9H +4C）

```
[[:接牌，"Shady"]，[:传牌，"Danny"]，[:抽牌]，[:抽牌]，[:抽牌]]
                                                  ^^^^^^^
```

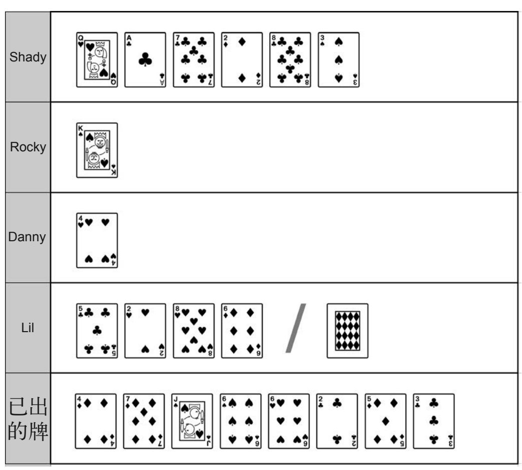
顺序有误（-8H:Danny +8C:Shady +4S +AS）

```
[[:传牌，"Danny"]，[:接牌，"Shady"]，[:抽牌]，[:抽牌]]
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

基于这些想法，你们俩共同实现了一个检查格式的函数，然后用相似的方法为该游戏记录构建了一个分析程序。在此过程中，Emma 学到了几个很有意思的文本处理技巧，但这不是重点。她迅速将注意力转回到手头的问题上。

4.4 善用演绎推理，检验工作质量

将格式错误的分支筛掉后，下一步就是从逻辑上判断，在剩下的 3 个格式正确的分支中，哪一个的出牌动作正确。解题进行到这里，画一个草图来表示每个玩家手中的牌再合适不过了，所以 Emma 马上画了一个。[2]



注 2：每个玩家手中的牌都是已知的，只有一个例外：Shady 传了一张牌给 Lil，但是不清楚是哪张牌。即便如此，那张牌也肯定是 Shady 一行中的某一张。

利用之前创建的出牌动作表，Emma 挨个检查了所有分支，边说边进行手动处理。

> Emma：第一个分支是 `Lil +8H:Shady -6D:出牌 -2H:Danny +JD +2D`。
>
> 我马上看出这是不可能的。因为 Lil 手中已经有了红桃 8，所以 Shady 不可能再传给她一张。
>
> **你**：没错！那下一个呢？
>
> Emma：第二个分支是 `Lil +8C:Shady -6D:出牌 -8C:Danny +JD +4S`。
>
> Shady 确实有梅花 8，所以有可能传给 Lil。Lil 也确实有方块 6，所以出这张牌也是有效动作。如果 Shady 真的将梅花 8 传给了 Lil，那么 Lil 绝对有可能立即将其传给 Danny，所以这里也没有问题。
>
> 最后，方块 J 和黑桃 4 都既没有出现在任何玩家的手里，也没有出现在已出牌堆里，所以 Lil 有可能抽到这张牌。我认为，这个分支是可能的。
>
> **你**：我觉得很正确。因为这个数据集只包含一轮游戏的情况，所以（理论上讲）我们不用看就能知道，第三个也就是最后一个分支是不可能的。不过为了检验我们的工作，无论如何都要验证一下。
>
> Emma：好的，第三个分支是 `Lil +8H:Shady -6D:出牌 -2H:Danny +7D +AS`。
>
> 我一眼就能看出，这个分支和第一个分支有同样的问题：Shady 不可能把红桃 8 传给 Lil，因为 Lil 手中已经有这张牌了。
>
> **你**：做得很好，看起来我们已经找到答案了。现在该放下练习数据集去迎接真正的挑战了。

作者的话

在设计以上这段对话时，我抄错了一个地方，导致我错误地认为第二个分支也是无效的。直到写下“无论如何都要验证一下”这句时，我才发现自己的错误。真尴尬，不过这个可笑的意外恰恰能说明本节的重点。

4.5　欲解复杂问题，先知简单情况

有了两个练习数据集做铺垫，Emma 已经有了很扎实的基础，可以着手解决主问题了。但最困难的部分是，如何将这些零散的想法串起来，处理一个每轮包含很多分支的完整游戏。

在继续之前，你问 Emma 对接下来的工作有什么计划。

你：能和我谈谈你接下来打算怎么做吗？

Emma：没问题。这个比较难的数据集基本上就是我们看过的两个样本数据集的组合，对吗？

你：没错。第一个数据集展示了游戏动作是什么样的，从那里就可以开始追踪游戏过程中每个玩家手里的牌了。

第二个数据集展示的是如何缩小范围，让你最终能找到唯一可能的动作。

Emma：好的。我应该把代码改写一下。每次轮到 Lil 出牌时，先把格式不对的分支去掉。

之后，每次取一个剩余分支，把它的动作序列运行一遍。如果可以运行完分支中列出的所有动作，又没有出现不可能的动作，则该分支就是正确的。

我不太明白的是，为什么这个比较难的数据集这么小呢？每轮只有 3 个可能的分支，而且游戏总共只有 5 轮，那不就是最多只需要遍历 15 个可能的分支吗？看起来真的很简单。

你：不是，实际上有 243 个可能的分支。若想知道为什么，去看看这个数据集中 Lil 第一轮的 3 个可能的分支吧。用咱们之前的剪枝方法，看看你会遇到什么困难。

Emma 重新审视了数据集中的前几行。

```
Shady +?? +?? +?? +??
Rocky +5S +QH +6H +JC
Danny +?? +?? +?? +??
Lil +7C +3S +8D +9H
Shady -4H:出牌 -??:Danny +??
Rocky +10D -10D:Danny +4S +2D -4S:出牌 -JC:Lil
Danny +??:Shady +10D:Rocky +?? +?? +?? -4D:出牌
Lil +JC:Rocky +?? -??:Shady +??
*  +JH -7C:Shady +10C
*  +JH -8D:Shady +9S
*  +JH -8D:Shady +10C
```

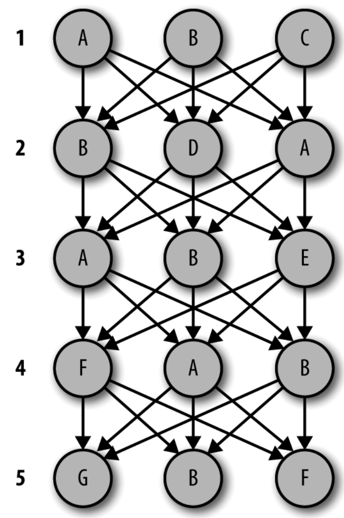
（……此后为后4轮数据……）

在纸上写写画画了一会儿，Emma 明白了你想让她注意的问题：如果不进行下一轮迭代，单从 Lil 的第一轮看无法排除任何一个分支。

当 Emma 继续往下看时，她才发现一个特定的分支可能会在以后的某一轮中产生不可能的动作，所以剪枝操作比她预想的要复杂得多。想要去掉一个无效分支，可能需要遍历游戏记录，这使得该数据集的手动处理过程比前两个要难得多。

搞清楚此题的这一细节后，还需要考虑很多问题。Emma 休息了片刻，以便消化这些内容。同时，你在考虑该怎么帮助她解决此题的这一棘手部分。

Emma 休息回来后，你给她看了你制作的该题的简化版，以便帮助她理解应该怎样进行后续工作。



Emma：哇，好多箭头啊！你给我看的是什么啊？

你：这基本上是一个理想化形式的“数扑克牌”问题。我知道它看起来有点抽象，但我保证它和解题思路有直接关系。

这个题中题包含 5 组字母。它的目标是从每组中抽出一个字母，最后生成 {A，B，C，D，E} 集合。取字母的顺序无关紧要，只要能把 5 个字母都取出就可以。

这些箭头代表的是，如果不剪枝而进行盲选，会出现的所有可能的选择序列。这样从上到下总共有 243 条路径。

Emma：啊，我明白你的意思了。5 组字母代表扑克牌游戏中的 5 轮，每组中的 3 种选择代表轮到 Lil 出牌时的 3 个可能的分支，243 其实就是所有分支的不同组合数。

明白了问题的结构确实很有用，但我还是有点不清楚怎样把这些转化为具体的实现细节。能再给我一点提示吗？

你：当然可以。假设你对这个图中的结构不甚了解，但很清楚目标是什么：获得一个（排序后）等于 {A，B，C，D，E} 的集合。这样一来，你就可以设计出一些规则，用以辨别无效分支。你能想出一个规则来吗？

Emma：我看到图中有其他字母，比如 F 和 G。如果分支中有这些字母，就说明它是无效的，应该直接排除。

你：没错，我们在游戏中排除结构错误的分支时，基本上也是这样做的。你还能找到其他更难察觉的限制条件吗？

Emma 想了几分钟，然后意识到，如果不小心从多组中选了同一个字母，那么也无法得到正确结果。比如，如果选了前 3 组的第一个字母，就会得到 ABA。因为接下来只剩两组可选了，所以就不可能得到完整的 {A，B，C，D，E} 集合。

你指出了这一观察结果和扑克牌游戏中“无效动作”的相似性。Emma 笑了笑，表示已经开始明白你的良苦用心了。

你：至此，我们已经把这个题目简化为了图遍历问题。我们用一个选择标准来确定无效分支，然后继续迭代，直到找到唯一的正确路径为止。

Emma：我能明白选择标准这一部分，但不太清楚怎么在不同的路径之间进行迭代，在这点上可能需要一点帮助。

你：数据集非常小，也许并不需要我们去找多么高级的启发式算法来缩小搜索空间。相反，可能只需要用简单的深度优先搜索就行了。

Emma：所以你的意思是：一直走最左边的路径，直到碰到一个无效分支，然后再往上跳一级，尝试下一个分支，由此不断重复？

你：没错！你现在试试看，告诉我前几个无效路径是什么样的。

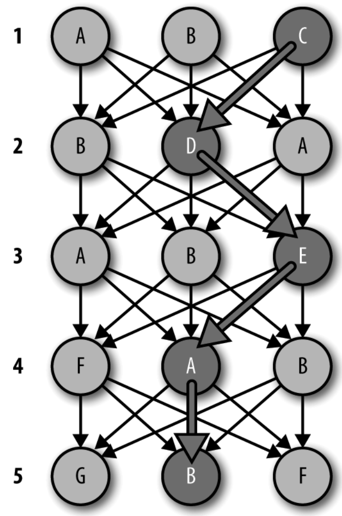
Emma：ABA、ABB、ABEF、ABEA、ABEB……我还要再继续找吗？

你：不用了。你已经成功排除了 AB 路径。下一步工作就是跳到 AD 路径，重复这个过程。

现在我想让你写一个程序，解决这个小问题。在你需要的时候，我可以帮你，但最好尽量自己完成。

写这个程序花了一些功夫，但 Emma 最终完成了代码的编写工作，程序运行起来了。她写的是一个脚本，对图进行递归遍历，尝试每一个路径。

Emma 运行了程序，然后在原图中描出了输出的路径，得到如下结果。



你：太棒了！你相当于成功完成了大海捞针的工作。我们现在是不是该回过头去解决原来的问题？

Emma：如果你不介意的话，我打算自己来解决余下的问题。你给我的帮助真是太有用了，我觉得自己可以把它应用到原题里，想办法解决。

你：嘿，这想法太好了。如果你需要帮助，就随时找我，但只要你感兴趣，自己独立解决问题肯定更有帮助。这也提醒了我，从我们今天的工作中，应该学到的最重要的一课其实是……

Emma：是什么？

你：真正的问题解决过程往往是孤独的历程。你可以从别人那里得到帮助，但最终还是要靠自己去理解所有的细节，这样才能解决问题。这也就是为什么无论何时都要做到"严密思考"：你需要非常精确、非常细致地理解问题。

Emma：非常有道理。面对很复杂的问题时，可以像我们今天这样把它拆分成简单的小问题并逐一解决，这样原来的问题就会迎刃而解。之前在遇到一个没法立即搞懂的问题时，我都像是撞了墙，感觉无从下手。现在我感觉自己能从不同的角度看待问题了。

你：听你这样说，我真的很高兴。不骗你，我们每个人都需要时时刻刻牢记这一点。不过看起来你已经有了正确的思路。那就祝你好运，去解决剩下的问题吧！

在收拾东西时，你看到 Emma 还在努力解题。这样一道难题，在之前她想都不敢想，而现在却引起了她的兴趣和注意，而且她极有可能解出这道题。

忠告与提醒

- 描述问题的原始资料一般都是零散的语句、示例和参考材料。为了理解这些资料，你需要记笔记，排除噪声信息，只留下最关键的细节。
- 每个问题的背后都有一堆简单的子问题，你早已知道如何解决它们。要将问题不断拆分，直到你能辨认出构成它的子问题为止。
- 难题由很多活动部分组成。先观察各个部分如何联系在一起，而不要管具体的实现细节。在写代码之前，先用纸笔解决部分问题。
- 在无效数据集上运用有效规则可能会得到难以调试的混乱结果。不要假设输入数据集的格式正确。在处理任何数据集之前，都要预先检查，以避免出现“垃圾进、垃圾出”的情况。

问题与练习

问题 1：从你自己的工作中，找出一个使用了严密解决方案的例子。在解决该问题之前，你被什么卡住了？最终你是怎样突破难点的？

问题 2：用极度严谨、接近方法论的方式去解决问题是有局限性的。在什么情况下，严密的思维过程反而可能会使问题更难解决？

练习 1：在遇到底层、严格且与上下文无关的工作时，严密的思维过程至关重要。请阅读“软件系统中的信息剖析”（http://pbpbook.com/anat）一文，了解这些概念如何用于文件格式和协议的设计。

练习 2：写一个程序，解决本章中的 {A，B，C，D，E} 问题。完成之后，再生成一个 26 层的新数据集，每层有 26 种选择，并且使其有且仅有一条路径能得到字母表中所有 26 个字母。如果不更改代码，你能处理这个新的数据集吗？如果不能，为什么？

你披荆斩棘，已经读完了半本书。万岁！

如果你还有兴趣，以下是另一个等待你解决的小谜题……

$	AF	$	DB	$	CE	$	8D	$	AA	$	87	$	37	$	B5
$	A2	$	87	$	64	$	37	$	6B	$	3D	$	1A	$	13
>	^	21	3F	AE	AC	C1	D2	24	15	#	66	>	^	00	00
00	#	68	>	^	D2	A9	10	16	41	00	00	00	AC	C1	D2
00	AC	C1	D2	24	00	#	67	>	^	00	00	00	00	00	?
00	?	21	29	25	24	00	AC	C1	D2	24	#	75	>	^	00
00	3F	AE	AC	00	00	21	29	25	24	00	00	3F	AE	AC	00
00	00	00	#	6E	>	^	00	#	20	>	^	00	00	00	00
00	?	D2	A9	10	16	41	00	00	00	00	00	AC	?	D2	24
00	00	00	00	21	3F	AE	AC	C1	D2	24	15	00	00	00	00
00	#	61	>	^	00	3F	AE	AC	00	#	79	>	^	00	00
00	D2	A9	10	16	41	#	69	>	^	00	00	00	#	3F	>
^	00	00	00	?	00	00	00	00	D2	A9	10	16	41	00	00
00	00	00	#	76	>	^	00	#	65	>	^	00	00	?	00
00	AC	C1	D2	24	3F	AE	AC	00	00	00	00	#	74	>	^
00	?	D2	A9	10	16	41	00	00	00	!	00	00	3F	AE	AC

行至栈（$tack）道中央，便是展开之时。秘密消息正藏于其间！

第 5 章

谨记自底向上，优化软件设计

假设你是一门软件设计课程的客座讲师，并且你希望缩小理论与实践的差距。

这门课程是你的朋友 Nasir 开设的，但是他目前的教学效果不太理想。于是，他请你来帮忙。

在进行案例分析的时候，Nasir 的学生很容易就能抓住要点，并能提出富有创意的问题，从而引出精彩的讨论。但是一旦涉及在自己的项目中运用设计概念，大部分学生都很难把理论和实践联系起来。

目前的问题是，大部分学生并没有构建软件系统的实战经验。这使得学生把软件设计看成是抽象的练习，而不是具象且必要的技能。

课本上的例子强化了自顶向下的软件设计方式，其中的设计理念出现得很突兀。真正的设计不同于此，但是学生经常照葫芦画瓢，进而产生气馁情绪。

为了揭示设计决策的来龙去脉，你将搭建一个小型的实时项目，并在此过程中与学生进行讨论。通过这样的形式，学生将有机会参与迭代设计过程，发挥主动性，一砖一瓦地完成系统的构建。

本章主要内容

你将逐步学会自底向上的软件设计方法，并权衡利弊。

5.1 找出关键词，认清问题

在第一堂课上，Nasir 简单地介绍了你将要搭建的系统：一个即时制生产工作流的小型仿真。

Nasir 没有用理论引入这个主题，而是描述了即时配送如何使网上购物变得更加可行。

- 当顾客从网上购买商品后，只要其住处与出货点的距离不超过 160 公里，货物一般都可以在一天内送达，最多也不超过两天。
- 地方仓库的库存量会在防止产品脱销的前提下保持最低水平。补货会持续进行：每当地方仓库向顾客卖出一件商品，紧接着就会有相应的订单发往更大的仓储中心，以便及时为此仓库补货。
- 仓储中心到地方仓库的货流是持续不断的，所以任何一个需要补充的物品可以立即被扔到卡车、飞机或火车上，前往目的地。
- 一旦货物从仓储中心运抵地方仓库，补货订单便会自动提交至第三方供应商。很多供应商会使用即时制生产工作流，这样就可以进行小批量的补货。
- 虽然整个订货流程从头至尾可能会需要几周的时间，但由于有效设置了货流，顾客能够从离其最近的地方仓库接收商品，而且仓库永远都有库存。同时，制造商提供的产品数量和实际卖出的数量大致相同。

在这一模式下，货物即时流向需要它们的地方，这样能使整个生产系统中的浪费和等待时间最小化。这种工作方式在现在已经很常见了，但在几十年前却被看作开创性的行业创新。

Nasir 花了一点时间进行介绍，然后示意你开始上课。为了跟上节奏，你给学生讲了一个小趣闻，以此引入今天所讲内容的几个细节。

> **你**：我父亲一生都工作在流水线上，见证了他所在的公司从大批量生产到即时制工作流的变迁过程。
>
> **学生**：那变化一定很大！好像是完全不同的两种工作方式。
>
> **你**：对，就是这样。公司在业务层面经历了巨大变迁，但生产层面几乎没什么变化。
>
> 公司转型之前，器件要一箱一箱地从上游供应方运过来，然后由工人进行加工，再被运往生产线下游。
>
> 当公司转型为即时制生产后，这一过程几乎和以前一样——只有一个很小的变化。工作流转向了：只有当空箱子从下游返回时，才会加工新的器件。

学生：所以换句话说，你父亲只有在下一个站点需要货物时才会开始工作？

你：没错！从单个站点看不出什么变化，但是整个系统从一开始最简单的部件到最后的成品都被串了起来。

以客户订单为准，从后往前进行生产。整条生产线只需要通过保持相邻站点一致，就能确定需要生产多少元件，以及何时生产。

这种过程深深吸引着我，因为它很有趣，看似简单的基础单元也具有实时性需要。出于这个原因，我觉得从无到有地对这种行为进行建模会很有意思，而且在此过程中我们也可以讨论一些有趣的软件设计原则。

Nasir 问学生是否通过刚才的故事理解了即时制生产工作流并已做好仿真的准备。学生尴尬地笑了，好像不知道他到底是在开玩笑还是认真的。但他随即又问了一个更清楚的问题：这个故事中的关键词是什么？

过了好几分钟，学生终于找出了和仿真相关的很多关键词，如器件（widget）、箱子（crate）、供应方（supplier）、订单（order），以及生产（produce）。

然后，你让学生从这些词中选出两个，组成一个比较容易实现的简单句子。沉思了一会儿后，其中一个学生喊出了他的建议：

“我知道了！我们来做一个箱子，然后往里面放一个器件！”

从这点着手很好，你谢过了那名学生，然后开始工作。

5.2 从实现最小化功能入手

开始演示时，你准备了几个极小的 UI 元素，包含的全都是简单的几何图形。你梳理了几个基本逻辑，同时学生看着你工作。

几分钟之内，你就在屏幕上做出一个小矩形，其中有一个圆形，初步代表“箱子中的器件”。

当你按下笔记本电脑上的空格键时，圆形消失了。再次按下时，圆形又出现了。唯恐学生不理解，你重复演示了好几遍……

再次集中注意力后，你列了一个图表，用以描述对象 `Crate` 的 API。

Crate

push(widget)

向箱子中添加器件

pop()

从箱子中移除器件

isFull()

如果无法再向箱子中添加器件，返回`true`，否则返回`false`

空箱　　满箱

这第一步就需要你做几个设计决策。尽管都是一些小事，但它们会影响到项目剩余部分的设计。

Nasir：我来简要概括一下到目前为止你所做的工作。现在系统中有两个对象：箱子和器件。箱子是可以装器件的容器，而且箱子是可以被装满的。器件还几乎没有被定义，我猜它可以用来表示任何产品吧？

你：没错。更进一步来说，我的建模对象是即时制生产系统中材料的流动情况，而真正被处理的内容并不重要。重点是这些箱子的情况，因为我们要确定是否需要生产新的材料。

学生：哦，我感觉有点明白了。你打算把这些箱子作为定制器件的信号，就和你父亲的工厂里的那些一样。

你：没错。现在来具体聊一聊这个功能。我们已经做好了箱子，并且可以查看是否需要补货。但是器件是我们凭空造出来的。这里缺了什么模块呢？

学生：某种供应来源？这是整个问题的重点，对吗？我们希望看到从箱子中移除器件的动作可以触发自动生产新器件的动作。所以，每一个箱子都要有一个供应方，并且供应方应该能够察觉箱子何时需要补货。

Nasir：听起来你是在暗示供应方应该监听箱子的状态，这不完全正确。不应该要求供应方主动检查箱子是否需要补货；相反，当器件从箱子中被移除时，供应方应该收到通知。

学生：为供应方加一个监听器，每当 pop() 被调用时就调用这个监听器，如何？

你：这些点子都非常好，但是有点太超前了。现在我们缩小范围，然后思考："好，我们已经收到一个补货请求了。这种情况需要哪些对象的协同参与？"

Nasir：这是个好主意。弄清楚系统中的事件流，与知道事件发生时需要做什么是两回事。我们一步一步地慢慢来吧。

学生开始发现，自底向上进行系统设计的一个难点是将对象之间的纽带解开，以便一小部分一小部分地进行实现，而不是一次实现一整块。这个技能非常重要，因为它有助于实现增量式设计。

* * *

你快速画了一张草图来展示填充箱子的过程。其中，你引入了对象 Order，用来将供应方与箱子联系起来。



一个学生问对象 Order 的意义是什么。如果让 Supplier 直接操作 Crate，不是更好吗？

这个问题问得很好，尤其是在项目的早期开发阶段。在设计中引入的任何对象都会增加概念包袱，所以无疑需要避免引入多余的对象。

但在此例中，如果不为 Order 建模，就很难区分系统的物理行为和逻辑行为。

在真实的生产车间里，上游的供应方直接将材料装入箱子，让人觉得好像 Crate 才是需要操作的对象。然而，箱子本身只是容器，其传达的信息不过是容量。

关于箱子目的地的真实信息可能存储在工人脑中，也可能打印在一张纸上或箱子外的标签上。这些信息就是 Order 所代表的内容。这个对象很容易被忽略，因为它并不像箱子中进进出出的材料那么明显，但无论如何它仍是模型的一部分。

总结完关于 Order 的全部问题之后，你开始实现填充箱子的工作流。过了一小会儿，你的仿真中又增加了一个三角形和一条线，而且你已经准备好要讲一堂基础几何课了。

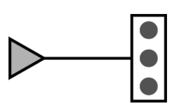


这些简单图形的含义远比其外表有意思得多。因此，尽管看起来简单，但它们却标志着项目有了实质性的进展。

你向学生解释说，当按下空格键时，方法 `order.submit()` 就会被调用，从而触发供应方生产器件。一旦生产完成，器件就会被送入目标箱子，从而完成订单。学生开始明白，这些基本单元最终将以某种方式组合在一起，产生更有趣的仿真模型。

5.3 避免对象间不必要的时间耦合

几天之后，你该进行第 2 次演示了。自从上次见面之后，你对仿真代码所做的唯一的大改动就是放大箱子的尺寸，这样就可以装更多的器件了。



这个很小但很重要的改动使你的模型可以支持软件设计中的 3 个至关重要的量：0、1 和“许多”。[1] 前面的例子只涉及前两种情况，但是从现在开始，你需要考虑所有情况。

由于已经建立了箱子填充机制，因此你需要实现的便是，一有物品从箱子里移出，就自动触发填充动作。你问学生如何实现这一功能，一名学生建议在调用 `crate.pop()` 后立即调用 `order.submit()`。

注 1：这被称为软件设计的 0–1–无穷规则（Zero-One-Infinity Rule），由 Willem van der Poel 提出。

于是，你按照学生的建议做了细微的变更，然后启动了仿真器。一个被装满的箱子出现了。你告诉学生，已经按照他们的建议设置好了空格键。随后，你按下空格键，屏幕上没有任何变化。你又按了一下，还是没有任何变化。然后你狂按键盘，屏幕闪了一下，但那个被装满的箱子仍然没有变化。

你加了几处日志记录代码，以确定仿真器能接收键盘输入，`crate.pop()` 和 `order.submit()` 被成功调用，以及没有产生死循环或递归调用等。看起来一切正常。你注释掉 `order.submit()` 那一行，又按了几下空格键，器件被一个一个地移除了。你从空箱子开始，注释掉 `crate.pop()` 调用，然后器件又一个一个地填满了箱子。

Nasir 问学生是否知道哪里出了问题。一名学生很快指出，对器件的移除操作和填充操作发生在同一帧里。因为两个动作之间没有间隔，所以箱子看起来没有任何变化。

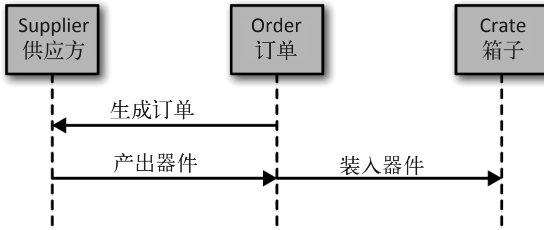
为了验证此猜想，你暂时给生产出的器件随机配了颜色。虽然演示结果乱七八糟，但它很好地证实了学生的猜想。

> **你**：现在我们已经知道哪里出了问题，那么怎样修复呢？
>
> **学生**：在产出新的器件之前，让 `Supplier` 暂停一秒，如何？
>
> **你**：这个想法很好，但我们现在的编程环境是异步的，所以并没有直接让进程休眠的方法。需要设置某种回调函数，令其在预设好的延迟之后执行。
>
> **学生**：好的，那就这样做吧。
>
> **你**：我会的，但是没那么容易。目前，调用 `order.submit()` 会立即触发对 `supplier.produce()` 的调用，后者会返回一个 `Widget` 对象。该对象随即会被送入 `Crate`。如果在 `supplier.produce()` 中使用异步的回调函数，就没法得到有效的返回值，这样整条供应链就会断掉。
>
> **Nasir**：所以现在的情况是典型的时间耦合。在 `Order`、`Supplier` 和 `Crate` 这几个对象之间，存在着时间依赖，这是由它们的设计方式导致的。如果要彻底解决这个问题，就需要重新设计，但现在暂时可以延迟订单提交进程，让它在系统接收到键盘输入之后一秒左右再运行。

你根据 Nasir 的建议做了修改，然后又试了一次。果然，刚一按下空格键，就有一个器件从箱子中消失了。过了大约一秒钟，箱子才被再次填充。随后，你连续快速移除 3 个器件，把箱子清空。过了一会儿，箱子又满了，而且所有新器件都几乎同时出现在箱子中。

看到系统正常工作，学生都很高兴。但你马上提醒他们，这样做治标不治本，其实有点投机取巧。为了使一切正常，需要改良工作流。

你绘制了一张顺序图，用来描述当有订单提交时，系统中的新事件流。



实现这个改进后的工作流并不需要对原系统做很大更改。

首先划分对象 `Order` 的责任，使提交订单和完成订单成为不同的事件。然后修改方法 `supplier.produce()`，允许它以回调函数的形式进行通信，而不是返回值。

在新的设计中，`order.submit()` 还是会立即调用 `supplier.produce()`，但现在是对象 `Supplier` 决定是否调用以及何时调用 `order.fulfill()`，从而完成对事件的处理。

Nasir 问了学生几个问题，以确定他们理解了这个小型的重构。很明显，学生能够正确理解执行过程，但仍不清楚做此更改的动机是什么。

你怀疑学生现在还没有理解新工作流如何生成灵活的定时模型。为了说明这一点，你快速实现了 3 种不同的 `supplier.produce()`。

1. 同步模型

   直接调用 `order.fulfill()`。可以立即填充器件，也就是像最初设计的那样。

2. 异步并发模型

   使用异步定时器，让 `order.fulfill()` 延迟一秒再运行，允许同时处理订单对象。

3. 异步时序模型

   将新到的所有订单对象全部放入队列，以每秒一个订单的速率相继进行处理。

上述各种实现方式的表现大相径庭，但都用了同一个 `Order` 接口。这证明已经去除了最初设计中的时间耦合，现在系统已经可以支持任何定时模型了。

全班简短地讨论了一下不同的定时模型以及它们各自的优缺点。

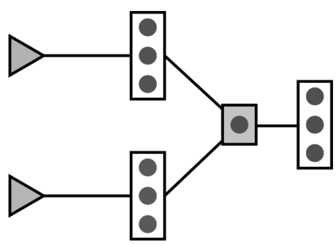
- 同步模型在逐步实现的仿真中很好用，因为一个事件循环在单位时间内只运行一次。但这样一来，要么需要放弃与系统的实时交互，要么得写一堆乱七八糟的代码鱼目混珠。
- 异步并发模型很有意思，但是如果不设计更复杂的 UI，那么用它很难说清楚订单的同步处理。
- 异步时序模型可以在其他可选项之间实现很好的平衡。它可以通过接受新订单，在整体上与系统进行实时交互。但是，器件在系统中的流通过程会随之产生连续且可预测的节奏。

你提出自己的建议：异步时序模型应该能在“有趣”和“易实现”之间找到很好的平衡点。学生也同意你的决定。如果这是一个有预设条件的真实项目，你可能没有条件自己做这个决定。但是，由于去掉了对象间的时间耦合，因此这个决定早晚还是要做的。

5.4 逐步提取可复用的组件与协议

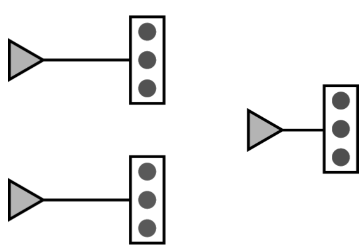
至此，你已经构建了供应方和箱子，并提出了按需填充箱子的机制。这些基本结构单元已经提供了运行即时制生产仿真器的大部分组件；剩下的任务只是构建一个“机器”(`Machine`)，既作为器件的消费者，也作为其生产者。

和学生仔细讨论了一些想法后，你决定，这个机器应该负责将两个输入源转换为一个联合的输出流。为了使每个人都参与思考，你做了一个图样，展示仿真器在添加这一新功能之后的样子。



Nasir 想让学生解释一下怎样实现这一新模型，但看起来他们都被这问题难住了。你思考了一会儿，寻找其中的原因。你发现学生的注意力都集中在寻找新系统和之前有什么不同上，所以他们看不到二者的相同点。

你降低了要求，让学生考虑如何使用他们已经熟悉的组件实现一个简化的系统。



你：这个例子有 3 个供应方和 3 个箱子。为了更容易理解，假设子系统是完全独立的。如果我们从其中任意一个箱子中移除一个器件，会发生什么呢？

学生：那样会触发提交一个填充订单。过一会儿后，供应方会完成订单，然后就会出现一个新器件。

你：很正确！现在我们对系统做一个小小的改变。假设最右边的这个供应方每次完成一个订单时，会消耗其左边每个箱子中的一个器件。这时会发生什么呢？

学生：左边的箱子就需要填充，所以订单会被自动送到它们的供应方那里。

你：完全正确。现在如果回头再去看之前的图样，就会更容易理解机器的工作方式。它和供应方一样会产出器件，但在此过程中，也会消耗上游箱子中的器件，继而触发向上游箱子的供应方提交订单。一个麻雀虽小五脏俱全的即时制生产工作流就这样产生了。

听了你的解释后，一名学生建议为对象 `Supplier` 建一个名为 `Machine` 的子类，这样就可以复用现在的 `Order`。你没有直接回应，而是请全班同学复查 `Supplier` 的实现，让他们自己总结并得出结论。

学生现在明白，对象 `Supplier` 的工作其实很简单：生成一个新器件，然后调用 `order.fulfill()` 完成填充操作。如果让 `Supplier` 立即完成订单，那么一行代码就可以实现。但是定时模型使问题变得有些复杂。

`Supplier` 自己有一段代码用来实现初步的异步时序工作队列。Nasir 很快指出可以复用这段代码，因为机器也需要实现延时订单处理，而且供应方已经实现了这部分功能。剩下的唯一问题是如何复用这段代码。

学生：如果创建一个子类会不会比较好？这样一来，`Machine` 和 `Supplier` 这两个对象就可以共享不少代码。

Nasir：咱们现在先不考虑二者之间有什么相同点，单纯考虑这个工作队列如何实现。它只是一个由函数构成的有序队列，这些函数逐个执行，间隔时间固定。那么这一过程对于 `Supplier` 的概念有什么特别之处呢？

学生：我觉得应该没什么特别的吧。你是说这只是一个实现细节问题吗？

Nasir：不完全是。我想说这是我们所用的工具链中缺失的一环。异步工作队列是极其普通的结构，但是因为我们用的语言没有内置这一结构，所以需要从头开始构建。

你：我刚开始就想到给工作队列创建自己的对象，但是之后意识到，如果再等一下，就会引出你们刚才的精彩对话。

Nasir：换句话说，你把选择权给了大家？够可以的啊！

虽然 Nasir 有些贫嘴，但是推迟决策确实是自底向上设计的重要部分。过早提取对象，然后尝试去想象未来的使用情况，可能导致接口变得乱七八糟；一旦考虑实际需求，就能更

轻松地进行接口设计。

再回到手头这个问题，你用了一点时间调整一些函数在代码库中的位置，然后为新建的对象 `Worker` 编写了 API 文档，如下所示。

Worker

delay
任务之间的等待时间

performAsync(job)
向工作队列中添加一个任务

run()
开始一个事件循环，按照delay定义的间隔时间逐个处理任务

重构之后，对象 `Supplier` 没剩多少代码了，所以不能把它当作基类。你从剩余部分中复制并粘贴了一些有用的代码，然后开始实现对象 `Machine`。

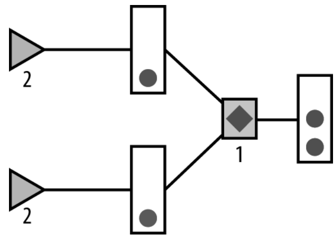
你首先加入了几个基本功能，使机器和上游箱子联系起来，这部分进展顺利。但此后的工作就变得有些复杂了：需要对 `Crate` 进行一些调整，以支持新添加的 `Machine` 结构。

你最终做的变更并不大，但很有代表性。当对象在与其设计初衷不同的场景中被复用时，经常需要这样的变更。

- 在只包含一个供应方和一个箱子的场景中，知道箱子是否空着并不重要。只要在器件从箱子中被移除时，能够立即提交填充订单即可。但机器只有在其上游箱子都装有器件时才能完成订单，所以你实现了方法 `crate.inStock()` 来获取此信息。
- 每个订单对象都会引用一个箱子对象，但反过来则不会。`Crate` 和与之对应的 `Order` 都已定义，因此系统在顶层运行良好。但在其中引入机器时，一切就变得乱七八糟了。为了让机器在消耗上游器件的同时提交填充订单，你用了一个涉及闭包的技巧，但解释起来既不简洁也不容易。[2]

你坦言，这种意外出现在对象连接点上的设计瑕疵，其实正是自底向上进行设计的缺点。但为了让大家重拾乐观情绪，你给学生展示了机器的一个可用版本，其订单数量可以实时更新。

注 2：对这个问题的正确处理方法应该是回过头，在对象 `Crate` 中给特定的 `Order` 加引用，但假设客座讲师的时间非常有限，你不想再去仔细考虑设计决策了。而这时出现了一种补救方法，可以掩盖种种细节，让学生能够将注意力集中在更重要的知识点上。



为了实现这个新特性，你只在系统内部做了一些小改动，并增加了几个帮助函数。除此之外，没有对 API 进行大的变更。这表示，整体设计到目前为止效果良好。

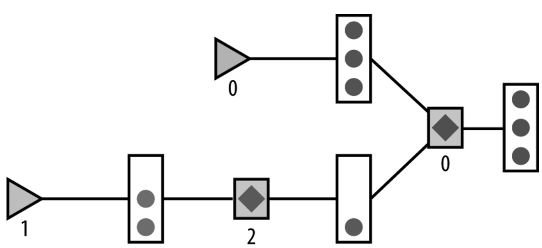
5.5 进行大量实验，发掘隐藏抽象

目前，工作中最困难的部分已经完成。Nasir 给了学生一些时间，让他们讨论如何对这个仿真做一些小的变更，以便测试此设计的优缺点。

开始时，学生提出的建议正如你所料，比如使不同的供应方和机器拥有不同的生产速度和箱子大小。看着系统根据瓶颈限制动态平衡工作量，大家觉得很有意思。他们对这些想法又继续探讨了一段时间，但结果并没有揭示任何与仿真设计相关的问题。

为了引导学生进行更有趣的讨论，Nasir 让他们实现一种新的机器。一名学生提议建立一个“纯化”模型：机器接收单个输入，并且产生单个输出，但在此过程中改变器件的类型。

Nasir 开始回应这名学生。但他还没说完，你就已经实现了这种新机器，并使它运转了起来。你将其输出放进一个“合并器”中，让这个例子更加生动。



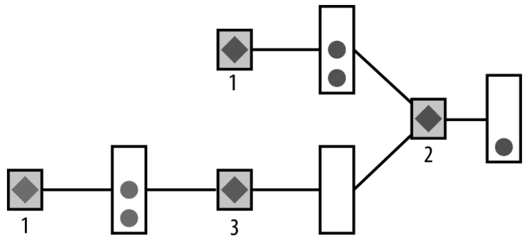
刚开始，Nasir 以为你已经预料到学生可能会问这种问题，所以提前写好了部分代码，但你很快指出并非如此。

实际上，这和你对合并器的定义有关：这种机器会从它的每一个供应箱中获取一个器件，然后输出一个器件。

根据这一定义，很容易实现纯化器（即只有一个供应箱的合并器）。因此，你可以很快实现这一新特性，而不用写任何新代码。

另一名学生甚至提出了更深一层的建议。他认为可以创建一种新机器，使其和对象 `Supplier` 的工作方式一样，即没有供应箱，因为任何集合与空集取并，其结果总为该集合本身。

这个提议令你很吃惊，因为你在创建对象 `Supplier` 时，从来没想过这个问题。但是果然，这个想法行得通！



学生们就这一主题提出了很多别的设想，包括在机器之间建立环形依赖、单一输入源为多个机器提供输入，等等。所有设想都如愿实现，虽然你在创建系统时并没有明确地针对这些用例做计划。以自底向上的方式进行设计的系统会有一些令人惊喜的性质。

Nasir 认为这节课该结束了，所以试着进行总结。他告诉学生，虽然这种实验很有趣，但只是为了帮助探索一些抽象概念。至于这些抽象概念是否能被正式支持，还要看它们是否能被证明有用。这种实验的目的并不是请大家去发现“隐藏特性”，然后不假思索地立即使用它。

学生似乎很好地理解了这一点。你对 Nasir 的提醒感到欣慰，因为你自己也时常忘记这一点。

5.6 了解自底向上方法的局限

这堂课的效果很好。你很高兴，开始收拾东西准备离开。这时，一名学生问是否能再问你一个问题。

> **学生**：这种利用域模型进行探索性实验的方式，可不可以用来识别设计缺陷？
>
> **你**：完全可以。如果想发现奇怪的极端用例，当然可以用这种方法来进行初步探索；但如果你在非极端用例中遇到问题，就需要及早注意了。
>
> **学生**：我举个例子吧。我们的机器可以很轻松地处理零输入、单一输入或许多输入。但到目前为止，我们只测试了单一输出的情况。如果构建一个“分割器”，也就是接收一个输入、生成两个输出的机器，会怎么样？
>
> **你**：嗯……这问题问得很好。我需要考虑一下。

几名学生在下课后留了下来，看着你用几分钟时间试着为仿真器增加一个分割器。你最终

实现了这种机器，但对代码的呈现方式不太满意。

你开始思考为什么实现这个特性比实现其他特性要难得多。这时你才发现，它们的基础结构不一样。合并器、纯化器和发生器[3]全部符合“多对一”的映射模式，即 n 个输入对应一个输出（3 种机器分别为 n= 许多、n=1 和 n=0）。但分割器是把 n 个输入映射为 n 个输出，这就改变了问题的本质。

分割器并不一定是不好的设计，但这说明，需要在自顶向下和自底向上之间权衡利弊。在自顶向下的设计方法中，你一开始就会考虑可支持的机器类型，然后再去创建满足这些类型的抽象层。这样做可以让系统变得更容易集成，但对于简单情况而言，可能会让代码过于复杂，因为需要提前做更多的规划。

你向学生解释道，在实际应用中，自底向上和自顶向下是相辅相成的。自底向上的设计方法有利于发掘新的设计理念，同时让设计保持简洁。当你遭遇死胡同或碰到难点时，自顶向下的设计方法可以帮助你从整体上思考问题，将事物之间的联系统一起来。两种设计方法并非水火不容，它们只是用途不同。

你感谢那名问分割器问题的学生，说他问了一个好问题。他朝你笑了笑。Nasir 草草记下一些笔记，以便给错过这节“福利课”的学生分享。但你知道，那些亲身参与了这节课的同学一定会感到很幸运。

注 3：也就是学生提到的没有供应箱的机器。——译者注

忠告与提醒

- 开始进行自底向上设计时，列出问题的关键词。然后，用这些词造出有意义的短句，越短越好。以这一短句作为指南，实现第一个特性。
- 在为项目增加新功能时，注意对象之间的联系。重视那些在量和时间上灵活的设计，这样对象就不会向其协作对象施加人为约束。
- 在提取可复用的对象和函数时，寻找不太随时间变化的基本结构单元，而不是去找那些表面上看起来可以复用的模板式代码。
- 要充分利用那些在复用基本结构单元的过程中意外出现的特性，以便解决新问题。但同时要注意，在拼接对象时，不要使代码过于复杂。如果集成点过于混乱，就说明不适合使用自底向上的设计方法。

问题与练习

问题 1：有些工作环境非常适合使用自底向上的设计方法，有些则不适合。在应用本章讨论的技术时，可能会遇到哪些非技术性的（也就是商业层面的）障碍？

问题 2：假设你决定构建自己的邮件客户端。你能找出哪些关键词？用这些关键词，可以造出什么短句，用以描述你要实现的第一个特性？

练习 1：用 20 ~ 30 分钟学习“共生性”（Connascence，http://connascence.io）这个概念，并记一些笔记，思考它与本章内容有何联系。

练习 2：回答问题 2，并接着做下去，为你的邮件客户端实现第一个简单却有意义的特性。尽可能缩小问题范围，一次性完成此练习。

第 6 章

认清现实瑕疵，改善数据建模

假设你在一家小公司工作。该公司刚着手对用了 10 年的考勤管理应用进行替换。

你的同事 Mateo 是原应用的主要开发人员。该应用在 10 年前替换了繁琐得惹人厌的纸质流程。多年来，该软件一直“任劳任怨”地工作着，但 10 年的连续使用使它暴露出一些弱点。其核心数据模型有一些粗糙之处，尤其不应该延续到新系统里。

由于你并没有参与原应用的开发，因此 Mateo 希望你能够从一个全新的角度思考替换方案。最近几天，你一直在研究原应用的代码库，并试验新想法。今天，你要向大家介绍构建更优系统的计划。

该项目最大的挑战是要解决理想情况下的技术实现与公司的工作风格和需求之间的矛盾。数据建模和工作流设计密不可分，只有把它们放在一起考虑，才能达到最好的效果。

Mateo 会帮助你了解该项目的历史。你们俩需要合作设计出一个能长期使用的新应用。

本章主要内容

你将学到，只要对数据模型的基础结构做很小的改动，就能从根本上改善用户与系统的交互方式。

6.1 分清概念建模和物理建模

你从理想情况入手，首先研究员工在普通的工作日里如何使用考勤管理系统。

- 上午 8:30：打卡上班；
- 下午 1:30：打卡吃午餐；
- 下午 2:30：午餐时间结束；
- 下午 5:15：打卡下班。

在原应用中，这一事件序列会创建一对工作记录对象 `WorkSession`，并在数据库中将它们建模为两个区间。一个是上午 8:30 到下午 1:30，另一个是下午 2:30 到下午 5:15。

你指出，这种设计在理想情况下是有意义的，但会给原始数据的处理引入不必要的麻烦。Mateo 让你举个例子，你很乐意这么做。

> **你**：假设一名员工忘了在上午 8:30 打卡，而其他 3 次都没有忘记。那么系统会怎样创建区间呢？
>
> **Mateo**：嗯……系统会误将下午 1:30 的打卡动作当作进入公司，因为这是该员工在一天中第一次打卡。从这里开始，系统会创建一个从下午 1:30 到下午 2:30 的区间，然后另一个区间从下午 5:15 开始，但结束时间未定义。
>
> **你**：正是如此！如果不修正这一错误，数据就不能与现实同步，并且会出现很奇怪的现象。更糟糕的是，如果想让数据恢复一致，需要修改两条记录中的 4 个域，非常麻烦。

为了阐明观点，你展示了一张包含两条工作记录的表，试图解释如何进行修改。

签到	离开	工作时长
~~下午1:30~~ 上午8:30	~~下午2:30~~ 下午1:30	~~1小时~~ 5小时
~~下午1:30~~ 下午2:30	~~空~~ 下午5:15	~~空~~ 2.75小时

你还提到，自己在系统中调查了忘记打卡这种事发生的频率。统计数据表明，这种令管理人员头疼的事每天都在发生。

Mateo 思考了一会儿，想到这是一个相对较新的问题。几年前，员工每天只需要打卡两次。在当时的工作流中，员工需要在上午 8:30 签到，并在下午 5:15 打卡离开。午餐时间是预设的，并会自动从员工的工作时长中扣除。通过这种方式，就可以按需修改工作时间段，而且修改起来不会特别麻烦。

系统创建 7 年之后，公司制度发生了改变，员工必须通过打卡严格记录所有休息时间。为了支持这一新规定，系统需要升级。但当时预算实在太紧，而且遗留代码库尘封已久。在这些限制下，根据新规定进行系统升级便成为了不可能的事情。

你已经注意到了这个设计缺陷，认为应当在新系统中予以修复，并且已经有了一个具体的解决方案。

> **你**：我的目标是，在修改错误的打卡数据时，不必修改正确的数据。如果需要添加缺失的打卡时间，应该能够直接键入，同时不修改其他数据。
>
> 为了实现这一功能，不能再将工作时间段建模为数据库中的区间，而应将每次打卡看作独立事件。这样就能在需要展示报告或运行计算时，从应用层把原始打卡数据转换为区间。
>
> **Mateo**：如果这样做，若某个打卡时间缺失，时间表中的签到 / 离开数据对仍然是错误的。
>
> **你**：话虽如此，但在原系统里可不只是出现错误报告这么简单——数据本身已被损坏。所以，仅因为系统将打卡数据放错了位置，就需要修改一大堆字段。
>
> 在新模型中，即使在某些极端情况下数据报告仍然与现实不符，但系统中的基本数据仍然是对的。无论员工何时打卡，打卡动作都确实发生了，所以系统会将其看作事实。由这些打卡数据生成的工作时间段是比较模糊的概念，但这种方法可以很清晰地区分打卡时间和打卡动作。
>
> **Mateo**：我懂了！你说得很有道理。不过，我还是想通过一个例子看看新模型怎么工作。

你很快又画了一张表，展示可以通过新模型直接添加缺失的打卡时间，而不必修改其他数据。

打卡时间
上午8:30
下午1:30
下午2:30
下午5:15

这样，你就可以在应用层为每对连续的打卡数据创建新的区间，从而将数据转入对象 `WorkSession`。由于这些区间会在运行时动态生成，因此在更新原始打卡数据时，不需要特别考虑。

在数据源混乱的系统中，一般最好保留一定的灵活性，不要在模型的物理层强加太多结构。

6.2 明确设计模型，追踪数据变化

盯着一堆 10 年前的代码看，你就只能猜出这么多了。于是，你向 Mateo 询问一些细节，让他告诉你原系统如何处理审查日志。他给你分享了一些背景资料，帮助你理解当初是什么需求推动了这一特性的实现。

- 从一开始，需求就很明显：软件需要对所有时间记录的变更进行全面的跟踪审核。该应用中的数据直接与员工的工资挂钩，也就是说可能出现不诚实的行为。
- 应时常核查员工的原始打卡数据，以寻找记录中的问题。比如，假设某员工总是忘记打卡，或总是要求将其打卡时间改为更早的时间，那么其中可能就有问题。[1]
- 审查需求来自于在出现真正的异常情况时保护员工的利益。在该应用的整个“服役期”，公司总共只审查了几次记录，而且事实证明假设是正确的。

为了降低成本，Mateo 使用了一个第三方库。该库提供的功能和备份机制差不多，只不过备份的是数据库记录。因此，只要旧系统中有记录被更新，就会产生如下工作流：

(1) 在修改之前，为记录创建一个只读副本；
(2) 根据需要更新记录；
(3) 更新字段 `admin_id`（管理员 ID），以此说明是谁批准了此次改动；
(4) 增加记录的版本号。

记录的副本被保存在其原始版本的表中，但拥有所有必要信息，以便对比各版本间的差异，或在必要时退回到旧版本。需要注意的是，由于以上操作都是在数据库层面上进行的，因此“修订”这一概念只针对插入或更新记录，而不针对实际的事务。

在讨论中，你一直在用“添加上午 8:30 漏掉的打卡记录”这个例子。为了展示版本机制如何适用于这种情形，Mateo 举了下面的例子。

注 1：往好的方面说，出现这样的情况可能意味着需要调查是什么导致该员工总是在签到之前开始工作。往坏的方面说，这种现象可能说明有人故意伪造自己的时间记录。无论是哪种情况，审查日志都是有帮助的。除了帮助发现问题，它还可以在日后作为证据，表明最初为什么会质疑这种现象。

打卡时间

上午8:30
下午1:30
下午2:30
下午5:15

不同版本的工作时间段

时间段ID	签到	离开	管理员ID	版本号
1001	上午 8:30	空	空	1
1001	下午 1:30	下午 2:30	空	2
1001	上午 8:30	下午 1:30	**1234**	**3**
—	—	—	—	—
1002	下午 5:15	空	空	1
1002	下午 2:30	下午 5:15	**1234**	**2**

Mateo 试着解释这种机制。他知道这种机制比较混乱，需要在新系统中加以改善。

> Mateo：为了添加一条遗漏的打卡记录，需要为两个工作时间段都创建新的版本。从数据中可以看出，这些变更是由经理完成的，因为操作者需要拥有管理员ID。
>
> **你**：但怎么表示这两个变更实际上来自于同一个变更需求？
>
> Mateo：没法表示，至少没法用数据直接表示。想要根据变更来推断发生了什么，必须把该员工在这一天中的所有记录全部调取出来。
>
> **你**：所以你的意思是，比如发现下午 2:30 在变更后既是时间段 1001 的离开时间，又是时间段 1002 的签到时间？
>
> Mateo：嗯……差不多吧，这真是超级混乱。有几次我需要根据这些数据生成报告。我真是费了半天劲儿才弄清楚怎么回事，然后才整理出一份清晰的报告，交给管理团队。这些数据本身就是一团糟。真是谢天谢地，我不需要经常处理这些数据。
>
> **你**：我能想象，事情很容易就会变得更糟。如果经理在输入时间时打错了字，后来才回过头去改正，会发生什么？即使在提交变更后马上改正错误，是不是也会创建一个新版本？

Mateo 点了点头，表示你说得没错。他说自己也发现了这个问题，并且非常希望你能提出全新的解决方案。

你开始描述自己的设计想法。其中，审查日志不是在数据库层面实现的一个额外的特性，而是被明确地建模为业务领域的一部分。

你知道，用一些样本数据可以帮助 Mateo 更好地理解这一模式，于是给他展示了下面的例子。

打卡时间

上午8:30
下午1:30
下午2:30
下午5:15

修订后的时间表

工作日	备注	管理员ID	ID
2016-03-17	忘记当天的首次打卡	1234	1001

调整后的打卡数据

操作	打卡时间	被修改记录的ID
添加	上午8:30	1001

你解释道，时间表修改模型 `TimesheetRevision` 表示与变更相关的高层信息：该记录属于哪个工作日，备注变更原因，指明批准该变更的管理员，等等。在此基础上，打卡数据调整模型 `PunchAdjustment` 就能捕捉到需要添加进时间表的打卡信息。

然后，你给 Mateo 展示了几个例子，解释新模型如何满足公司常有的几个其他类型的变更需求。

某员工忘记签到，直到晨会后才去打卡。

打卡时间

上午8:30
~~上午9:17~~
下午1:30
下午2:30
下午5:15

修订后的时间表

工作日	备注	管理员ID	ID
2016-03-17	按时上班但晚打卡	1234	1001

调整后的打卡数据

操作	打卡时间	被修改记录的ID
添加	上午8:30	1001
删除	上午9:17	1001

某员工忘记在午餐时间打卡。

打卡时间

上午8:30
下午1:30
下午2:30
下午5:15

修订后的时间表

工作日	备注	管理员ID	ID
2016-03-17	忘记记录午餐时间	1234	1001

调整后的打卡数据

操作	打卡时间	被修改记录的ID
添加	下午1:30	1001
添加	下午2:30	1001

这些例子说明了新模型的一些优点，但是 Mateo 还想问你几个问题。

> Mateo：总的来说，我同意这个方法可以使审查跟踪更容易理解。但它对我们有什么其他用处吗?
>
> **你**：说实话，我采用这种建模方法的初衷是整理审查系统，但随后意识到，它可以为管理员提供更好的工作流。
>
> Mateo：为什么呢? 从你展示的这些内容里，我还看不出来这一点。
>
> **你**：在现有的系统里，`WorkSession` 记录是被直接编辑的，而你所使用的审查工具在任何更改发生之前都会为其创建一个只读副本。
>
> 但如果一次变更同时编辑好几条 `WorkSession` 记录，就很难把它们联系起来。这限制了我们实现使编辑过程的容错性更高的特性（至少是使特性变得更复杂）。
>
> Mateo：能说得更具体一些吗? 我的思维可能相当闭塞，因为我一直站在工作了 10 年的老系统的角度考虑问题。
>
> **你**：当然可以！如果你在正式更新记录之前，能审查一下即将做出的变更，是不是很好?
>
> 如果用模型 `TimesheetRevision` 生成实时预览，就能在提交变更之前改正所有错误。
>
> Mateo：嗯……没错！那很有用。我现在明白你为什么这样建模了：你打算用 `TimesheetRevision` 和 `PunchAdjustment` 驱动打卡记录的变更，而不是用相反的方式。
>
> **你**：完全正确。我打算大致模仿事件溯源模式。[2] 通过将我们想要对时间表做的变更表示为一个 `PunchAdjustment` 事件序列，可以推迟更新原始打卡数据。

Mateo 对事件溯源模式还不甚了解，于是他问如果前后数据不一致会怎么样。你这样建模的目的就是从根本上解决这个问题。

事件溯源模式将独立事件建模为不变的数据。这些数据代表不变的事实。通过遍历事件序列并计算结果，可以看到系统当前的状态。由于数据在以事件为基础的模型中单向流动，因此这一状态将永远与生成它的事件序列保持一致。

对于每条打卡记录，其整个生命周期非常直观。打卡记录的创建方式只可能是下述两种方式之一：员工自己打卡，或者经理批准调整打卡数据。

注 2：事件溯源模式（http://pbpbook.com/event）使得对数据集的变更明确、可逆且可审查。

无论通过何种方式创建，打卡记录一旦生成，其时间戳就永远不会改变。对于已创建的打卡记录，唯一可能发生的变更是被标记为已删除。要删除记录，只能通过经理批准调整打卡数据。打卡记录一旦被删除，就不能再进行交互了。

你向 Mateo 指出，新系统中的打卡记录只有两个状态：已创建和已删除。而且，由于每个 `TimesheetRevision` 都代表对打卡记录的一组连续变更，因此可以知晓哪些记录被修改过以及为何被修改。[3]

Mateo 想了一会儿，然后又问了一个问题。

> **Mateo**：这个想法听起来不错，但如果几个 `TimesheetRevision` 请求出现冲突，该怎么处理呢？如果同时出现两个请求，事情会变得很麻烦。比如，一个请求添加一条打卡记录，另一个请求删除一条打卡记录，而对二者的处理是分别进行的。
>
> **你**：这是一个好问题。如果对某个时间表同时进行多处修改，情况会相当复杂，而且可能会使前后数据不一致。另外，我们绝对不想处理多路合并的状况！
>
> 为了避免这些麻烦，我们可以为系统添加限制，使得在任何时间点上每个员工 / 工作日组合拥有的处于开放状态的 `TimesheetRevision` 都不能超过一个。可以不经批准对处于开放状态的 `TimesheetRevision` 进行添加操作或删除操作，但呈现出的整体效果就是当天的一个连贯的时间表，并且它处于挂起状态。
>
> **Mateo**：好的。这种限制的实际效果还有待观察，但就目前来看还是可以接受的。

这场讨论即将结束。你注意到，新的设计思路似乎沿着一个主题展开：通过将可变状态最小化，尽可能减少偶发的复杂事件。

关于这个主题，还有很多值得讨论的地方，[4] 但你早就急不可耐，想要讨论自己的下一个点子了。

6.3 理解康威定律，实践数据管理

> 组织在设计系统时会被其自身的沟通结构所限制，设计出的系统会有相同的结构。
>
> ——梅尔文 · 康威（Melvin Conway）

注 3：该问题很适合采用事件溯源模式，因为其可能的状态变化很少。更复杂的模型需要复杂的数据库查询，而且性能是你在选择模型时需要考虑的因素。

注 4：如果有兴趣了解可变状态如何使程序的复杂度激增，请阅读 Ben Moseley 的论文 *Out of the Tar Pit*。

你问 Mateo，公司目前怎样处理时间表变更请求。他立即意识到，这可能是当前工作流最大的缺点。

这个工作流是专门设置的。每个请求变更时间表的员工可以联系经理，并且可以用他们觉得方便的任何形式，如直接面谈、发邮件或打电话等。经理在审查这些请求后进行汇总，然后再发给工资管理员，由工资管理员在考勤管理系统内进行更改。

这一过程的反馈环灵活可变，但常常效率很低。要确认一个变更，通常会耗上好几天。发工资的前几天会变得很忙碌，因为需要使所有时间表的数据一致，以便正确结算工资。如果某个请求丢失，或请求的某些细节信息错误，可能需要在反馈环中转好几圈，才能更正。

针对这个有问题的工作流，员工想出了一个解决方案：如果使用考勤管理系统内置的即时消息功能直接向工资管理员提交变更请求，似乎可以更快地得到回应，而且修改得更加准确。但这样一来，经理就成了"局外人"。从管理层的角度来说，这并不是最优的解决方案。有些员工会同时向经理和工资管理员提交变更请求，以为这样能在不违反规定的情况下尽快得到回应，但其实会让系统更混乱。

公司的规模较大，这种混乱的情况导致每天摩擦不断，但还没有糟糕到需要公司立即加以解决。不过很明显，如果解决方案的成本不高，公司还是很乐意改进系统的。你的观点是，在新的考勤管理系统中很容易解决这个问题，因为新的数据模型能够实现旧系统不能实现的功能。

> **你**：我知道这可能让人难以接受，但我认为解决这个问题的根本方法是让员工自己调整时间表。
>
> **Mateo**：我刚刚就怕你会这么说。我也觉得这个想法很好，但是恐怕很难说服管理层。我根本不知道该怎么说，因为这种做法和公司的工作风格实在相差太大了。
>
> **你**：那么你认为主要的困难是什么呢？管理层最担心什么问题？
>
> **Mateo**：首先，我认为他们会担心技术培训问题。他们之所以让工资管理员处理所有变更，而不是让各个部门的经理去做，有一部分原因就是早先在培训管理人员编辑时间表时效果并不好。
>
> **你**：别怪我说话不好听，但你不觉得出现这种情况，与系统设计得比较烂有一定关系吗？就为了向表中添加一条打卡记录，要修改 4 个文本字段，这即使对于程

序员来说也不方便。

旧系统无法审查变更历史，无法方便地撤销变更，也无法一次性修改一整天的时间。你能看到的，只是编辑的每个时间段的独立形式。产生这种结果的原因就是系统使用了自动生成的管理面板，而没有构建自己的用户界面。

Mateo：所以你是说，不友好的用户界面是罪魁祸首？如果 10 年前你跟我说这些，我不确定自己会同意；但现在，我的看法已经彻底改变了。但是这种“以人为本”的系统设计理念也就是近几年才兴起，即使在商业应用中也是如此。

也就是说，我们公司还有大量仍在服役的软件是 20 年前甚至更早时构建的，这些软件比考勤管理系统还要难用。这使整个公司都对软件敬而远之。所以，即使能够让管理层相信我们能构建出易学易用的软件，我们也需要超出他们的预期，才能获得批准。

你：他们还关心什么其他问题吗？如果能了解他们关心的问题，我们就可以想办法在提议改变工作流时，强调他们关心的那些方面。

Mateo：据我所知，管理层非常在意准确度。为了准确，人仰马翻也在所不辞。他们很担心由于时间表不准确而给员工少发或多发工资。这样的担忧是合理的。

他们的观点是，如果让工资管理员处理所有的变更，那么确保记录准确的责任就可以由工资管理员一人承担。工资管理员接受过良好的培训，熟知员工可能会犯的所有常见的错误，所以在遇到有问题的变更请求时能驾轻就熟。

你：你对此怎么看？这种方法的实际效果和他们想的一样吗？

Mateo：考虑到现有系统的局限性，我认为这种方法的效果还不错。主要的问题是工资管理员一个人需要承担的工作量太大了，而且我还不清楚这样做的成本效益如何。

你：没关系，我有办法处理这些问题。先不具体谈如何做，你认为还有什么需要注意的问题吗？

Mateo：他们还经常提到的一个重要问题是，急需有效的审查机制。即使是很不起眼的矛盾之处也值得深挖，以便提前预防欺骗和滥用职权的行为。

这种做法的弊端在于，它可能导致员工与公司相互不信任，而且会耽误经理的时间，让他们没有足够时间去做更重要的事。

在仔细思考影响公司运作的文化因素时，你开始明白应该谨记哪些设计约束条件。

你意识到，如果想让管理层至少考虑一下改进时间表编辑工作流的提议，你的方案必须简明易用，并且容易发现和修改数据输入错误。而且，需要保留甚至扩展现有系统的审查机制。

你相信自己设计的工作流不仅能满足上述所有需求，还能满足之后的其他需求。Mateo 看起来有些怀疑，但对你的建议感到很兴奋。

6.4 谨记工作流设计与数据建模密不可分

Mateo 也认为，只要采用正确的方式，允许员工修改自己的时间表将极大地改善工作流。

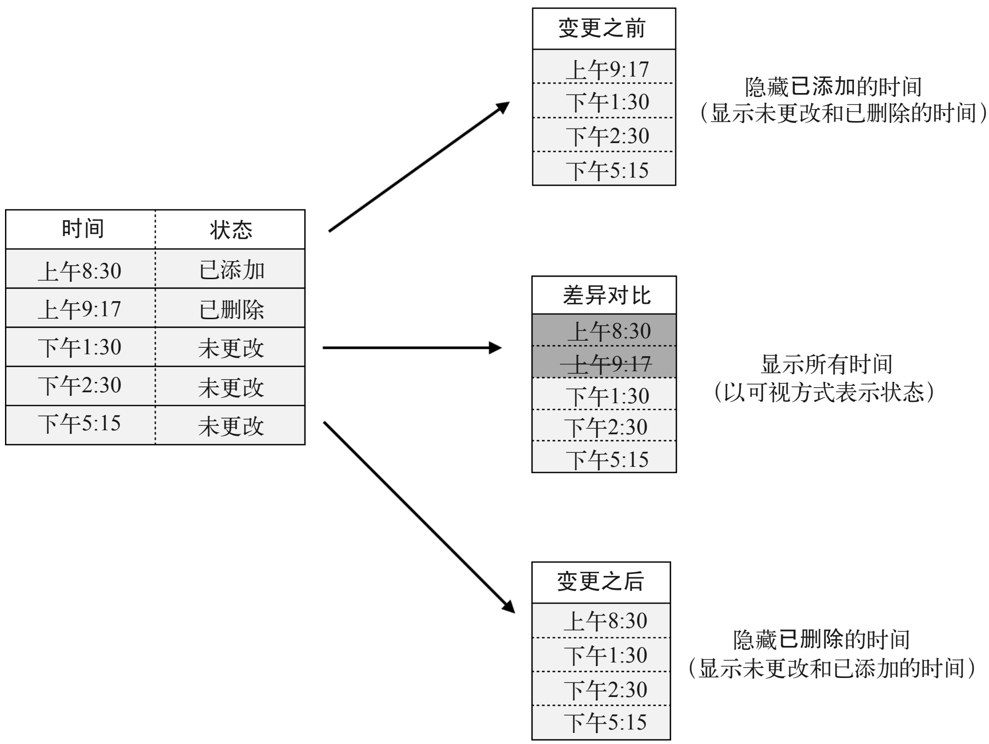
为了更清楚明了，你指出新的工作方式有几个具体的好处。

- 只要员工能够在提交前预览自己对时间表所做的变更，那么他们就能清楚地知道自己的时间表在变更请求被批准后会是什么样子。这可以避免由沟通不善引发的数据输入错误。
- 假设清晰地标注了所有挂起的变更，时间表和其他报告可以立即根据请求的变更进行更新，而不是继续显示不完整或不准确的信息。
- 不再依靠分散在五六个办公地点的多位经理汇总变更请求并提交给工资管理员，所有变更请求都将由员工自己直接输入系统，唯一需要保留的步骤是审查和批准变更。这样做可以减轻管理人员的负担，使他们少做些容易犯错的繁杂工作。
- 如果对某个变更有疑问，所有的管理人员以及提交此变更请求的员工，都会在同一时间看到同样的信息。如果需要更改变更请求，那么更改请求也会被实时更新到系统，以供任何有需要的人查看。
- 由于新的变更请求系统会将修改时间表数据的正式请求自行移入考勤管理应用，因此书面记录远比公司现有的记录完整和一致。
- 可以实现自动通知变更被接受或被拒绝，防止出现决策未通知到位的情况。
- 如果在工资结算期结束时仍有未处理的时间表变更请求，系统会给工资管理员发送提醒信息。

所有这些潜在的好处有赖于新系统的正确实现，以及系统处理变更冲突的能力。你已经针对新系统的风险和不确定性做了一番研究，并且草拟了几个图样，现在是时候展示解决方案了。

你的解决方案的核心是一个展示层对象，其中包含两条关键数据：某工作日的所有已提交的打卡数据，以及计划对此数据做的调整。

这个总的数据集将用于展示 3 条重要信息：变更之前的时间表、变更之后的时间表，以及所做变更的大致情况。

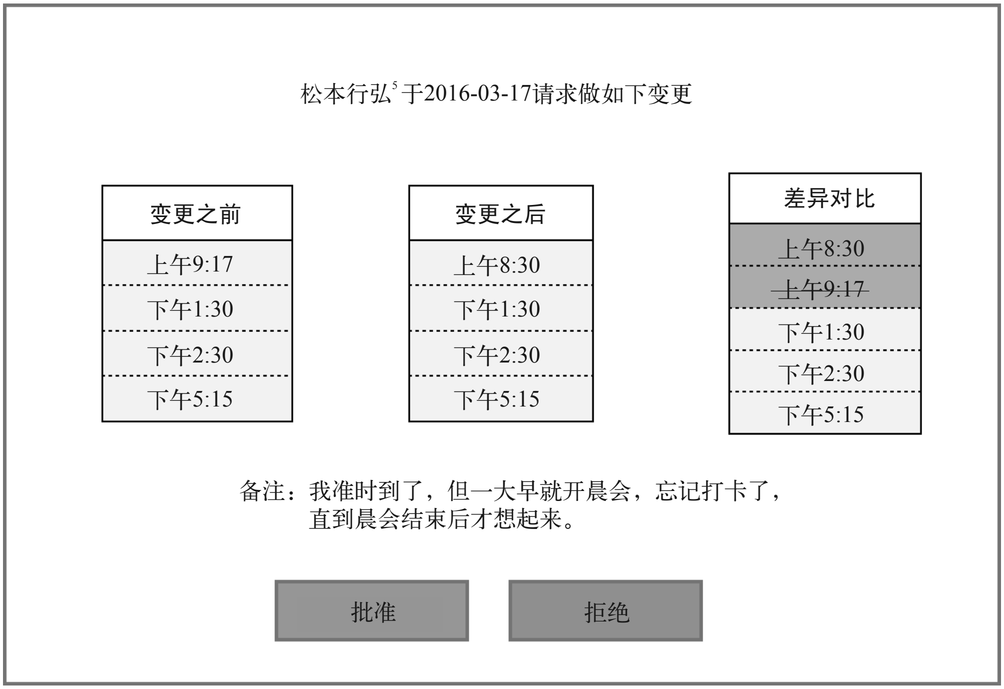


你指出，标有“变更之后”的视图并不只用于预览变更请求的最终状态；当 `PunchAdjustment` 被添加到 `TimesheetRevision` 中时，这个视图还可以实时更新。这样一来，就可以在用户界面中模拟编辑打卡数据的操作。



当员工准备提交变更请求时，系统会并排展示变更之前、变更之后及差异对比的视图。在检查并确定自己的变更准确无误之后，员工便可以填写“备注”一栏，解释为什么要申请变更。

变更请求一旦被提交，就会出现在管理面板上，如下所示。



如果请求被批准，那么系统将创建一条上午 8:30 的打卡数据，上午 9:17 的打卡数据则会被标注为“已删除”。如果请求被拒绝，系统将关闭 `TimesheetRevision`，打卡数据也不会有任何改变。在两种情况下，时间表都会回到没有挂起变更的状态。

该工作流最关键的特性是，用于计算员工工资的正式时间表只有在经过管理人员批准后才会被修改。这样就实现了与旧系统相同的集中控制和审查机制，但简化了沟通过程，从而大大减少了数据输入错误。

回顾你针对核心数据模型提出的改进方案，能清楚地看到，每个细微的改动都会促使系统进一步改进。令你感到惊讶的是，新设计并不需要原有模型经历翻天覆地的变化；只需要做较小的改动，使数据在系统中以一种新的方式存储和交互即可。

你不能保证管理层一定会接受新的工作流。在实际操作中，总会有公司规定和预算等方面的限制。即便如此，你还是对其中至少一部分想法抱有信心。它们终将通过新系统为公司的所有人提供便利。

注 5：松本行弘是 Ruby 语言之父。本书作者最常用的编程语言就是 Ruby。——编者注

忠告与提醒

- 保留数据的原始格式，不要试图立即将其转换为与概念一一对应的结构。你当然可以把原始数据处理成任何格式，但从复杂模型中提取与原始数据相同的信息，会带来不必要的麻烦。
- 在开发数据模型时，仔细考虑数据将被如何表示、查询和修改。在实践中，很少有项目只需要针对数据进行创建、读取、更新和删除等简单操作，因此一定要量体裁衣。
- 要把预览、备注、批准、审查和撤销事务性数据变更等操作设计得简单易行。要做到这一点，需要另外写代码，而不是依赖于预建的库。另外，在数据模型中采用事件溯源模式可以简化工作。
- 数据管理工作流的设计要尊重和支持软件使用者的组织文化。若在设计时不考虑康威定律，系统很可能会被成千上万种使用方法压得不堪重负，最终崩溃。

问题与练习

问题 1：本章所描述的考勤管理工作流非常适合拥有数十名员工和若干办公地点的公司。如果一家公司只有 5 名员工，且他们都在同一地点办公，应该怎样设计系统？对于拥有 5000 名员工和 50 个办公地点的大公司来说，又该如何设计？

问题 2：你所在的公司使用及制作的软件是否与公司的文化和沟通方式相契合？如果不契合，这种业务与软件之间的不协调会带来什么不良影响？

练习 1：找一个最近参与的软件项目，思考当出现人为错误时，项目数据会如何变得与事实不符。研究你的项目目前怎样处理这种情况。记下你发现的所有闪光点和缺点。

练习 2：使用事件溯源模式，为一个简单的井字棋游戏建模，并使其能够保存、还原和撤销每一步操作，而且能一步步地回放游戏。如果你还想做得更深入一些，再在游戏记录中加入分支点。

第 7 章

逐渐改善流程，合理安排时间

假设你是咨询师，专门帮助处于产品开发初期阶段的公司克服困难。

你最新的客户在几个月前刚从 Web 开发转型做产品开发。其核心关注点是一款叫作 TagSail 的产品。这是一个非常适合于移动端用户的 Web 应用，专门帮助用户了解附近的二手物品出售活动。[1]

TagSail 的商业模式很简单：它免费向买家提供服务，但向在其平台上发布销售信息的卖家收费。它还为付费用户提供了一些额外的功能。但和很多不成熟的产品一样，TagSail 所提供的功能有点老套。

几个月以来，该产品表现平平。但就在最近几周，它开始获得关注。这给公司的技术设备和人员都带来了压力。团队现在蓄势待发，力求不被市场淘汰。

你的任务是帮助 TagSail 团队“精兵简政”，同时仍能为用户提供稳定的服务。为了实现这一目标，你将运用精益方法改善流程，但会做一些调整，以满足客户的实际需求。

本章主要内容

你将了解到使软件项目管理陷入困境的一些反面模式，并学到如何通过逐渐改善各个层级的流程脱离困境。

注 1：在美国，人们经常利用自家的庭院或车库举办售卖活动，以处理自己不再需要的旧物。这种活动被称为 yard sale 或 garage sale。——编者注

7.1 敏捷、安全地应对意外故障

第一天，你就遇到了小的紧急事件。一个逆地理编码 API 出了问题，导致所有用户在打开 TagSail 的首页时都遇到了内部服务器错误。

你询问公司的开发主管 Erica，希望了解一些细节。

> **你**：我知道你现在不方便聊太长时间，但你可不可以用一分钟讲讲究竟发生了什么？
>
> **Erica**：当然可以。今天早晨，我们的访问量突增，因为某个很流行的订阅号提到了我们，这使得页面的加载时间明显变长了。为了满足访问需求，我们增加了服务器实例。这样做确实有一定的效果。但就在几分钟之前，我们的逆地理编码服务开始拒绝所有请求，使首页彻底崩溃了。
>
> **你**：所以此时此刻，根本没有人可以使用这个应用，对吗？
>
> **Erica**：没错。用户会看到一条写着“对不起，出错了”的消息，也就是发生内部服务器错误时都会出现的消息。这种情况真是糟糕，因为今天的单日访问量是我们有史以来最高的。
>
> **你**：你觉得还需要多久才能让应用恢复正常？
>
> **Erica**：还不确定。我们还在努力寻找逆地理编码 API 出问题的真正原因以及修复方法。我们认为原因可能与流量限制有关。

你继续观察了几分钟，然后提示说，团队的关注点可能错了。团队应该集中精力让应用恢复正常，而不是想办法修复出问题的 API（即使这样做可能会稍微影响功能）。

在简短的讨论之后，开发人员开始意识到，逆地理编码 API 并不是关键组件。检测访问者的地理坐标以及根据坐标显示地图，这些工作是由另一组 API 完成的；只有在将坐标数据转化为地点名称，并在地图上方的搜索框中显示时才需要逆地理编码服务。

暂时停止对逆地理编码 API 的调用，会使搜索框没有内容。这种情况在定位不准确时可能会给访问者带来小麻烦，因为他们会一眼看出地图并未根据自己的位置显示。即便如此，只要在搜索框中手动输入正确地点，访问者仍然可以正常使用应用。

尽管大部分团队成员都同意这一想法，团队中最有经验的前端开发人员 Sam 仍然有异议。他表示，将服务器端的逆地理编码 API 调用移到客户端，也许可以彻底解决这个问题。这样做既可以消除流量限制，又可以完全恢复应用的功能。你与 Sam 和 Erica 简短地讨论了一下这样做的利弊。

你：Sam，你是已经实现了客户端，还是打算现写代码？

Sam：这个嘛，当初实现这个特性时，我就建议这样做，而且还研究了一番。我不知道还能不能找到那些代码，但照着文档做应该很容易实现。

你：如果选择这样做，你认为多久能实现变更？

Sam：我认为很快就能完成，最多半小时吧。

你：你当时研究时，用的环境有多真实？有没有模拟过同时收到大量请求的情况？有没有在产品需要支持的所有浏览器中测试过？有没有在真实环境下处理过流量？

Sam：没有。但是这个 API 是由 FancyMappingService 提供的。因为这种用法很常见，而且 FancyMappingService 又这么流行，所以我想应该没问题吧。

你：我想你也许是对的。但我很担心压力之下的实验效果会不好。如果先暂停使用这个特性，让应用重新可用，那么就不会有压力，大家的思路也会更清晰。

Erica：咱们都退一步怎么样？ Sam 可以现在就着手编写把逆地理编码 API 调用移到客户端的补丁，同时我会让应用暂停使用这个特性，为大家争取时间。这只需要几分钟，大不了撤销操作，回到现在的状态。

你：听起来不错，只要你能等到系统重新稳定下来后再尝试打补丁就行。

Erica 顺利地停用了逆地理编码特性。她问你现在立即部署修复代码是否合适。你指出，即使是在现在这样一团糟的情况下，仍然不要急功近利，最好快速复查一下，以免雪上加霜。

Erica 为 Sam 创建了一个拉取请求，供他复查时用。然后，她和你绕着办公室走了一圈，看看大家都干得怎么样。

经过漫长的等待后，Erica 终于收到 Sam 通过聊天窗口发来的消息。他还在忙着编写补丁，并且他认为再等 15 ～ 20 分钟就可以发布。此外，他想跳过当前的应急步骤，直接部署自己的修复代码。

你什么都没说，但从你的表情可以明显看出，你不认可这种做法。你穿过大厅进了 Sam 的办公室，并且关上了门。

5 分钟后，Erica 收到一条提示信息，显示她的分支已经部署了。Sam 随即跟着你回到 Erica 的办公室。Erica 打开服务器日志，你们三人一起监测系统。

请求日志在屏幕上一行行地滚动着，这说明人们已经可以重新加载首页了。不出所料，手

动搜索地点的人显著增加。

确定系统重新稳定下来之后，Erica 让 Sam 继续编写客户端补丁。眼下压力得以缓解，不再需要着急部署修复代码，而是可以仔细复查、认真测试。

7.2 识别并分析操作瓶颈

自从你上次到访，已经过去一周了。再次见到 Erica 时，你问她的第一个问题就是近几天有没有发布什么新特性。

她告诉你："没有，除非把 bug 修复也算上。"你清晰地看到，她的眼中闪过一丝失望。你没有浪费时间，直接进入了工作状态。

> **你**：如果上周没有发布任何新特性，那么团队里的人都在忙什么呢？
>
> **Erica**：我想想……我开始将应用和几个分类广告网络做整合。
>
> Sam 忙着给以前建的内部库制作新版本，这是为下个月要实现的几个新特性做准备。
>
> Sangeeta 和 David 正在做一项改进，我们本来打算这周发布的，但后来出现了几个紧急情况，需要他们提供支持。于是，他们只能暂停手头的工作，去处理紧急情况了。
>
> **你**：什么紧急情况？
>
> **Erica**：和分类广告整合也有关系。几周前，我们增加了对一个很流行的广告网络的支持。一开始，它看上去一切正常。但我们后来发现，新版本的 API 仅支持特定地区，其他地区只能使用旧的 API。
>
> 两个 API 之间的差别很小，我们以为只要不使用新特性，它们可以共享同一个客户端。但事实证明我们错了。
>
> **你**：那么你们是怎么发现这个问题的？
>
> **Erica**：从用户的 bug 报告里发现的。针对整合，我们还没有良好的监管机制，所以只能依靠客户支持团队。
>
> 如果某个 bug 只出现了一两次，我们就认为它只是特例。我们每周都会复查 bug 报告，并按照优先级排序。如果同一个 bug 出现在 3 个或更多的报告中，就会引起重视，我们会立即派人修复。这次就是这样的情况，而且这次的 bug 耗费了 Sangeeta 和 David 半周的时间。

你：他们解决问题了吗？

Erica：我们认为解决了。对于好几个与我们集成的系统，我们没有直接访问权限，特别是没有为所支持的每个系统的每个可能版本都搭建预发布环境。Sangeeta 和 David 的修复工作看起来已经解决了 bug 报告中的问题，但是否彻底解决了兼容性问题还比较难说。

你：总的来说，这种情况很糟糕啊。

Erica：没错！我感觉我们至少花了半周时间去处理这些整合问题，但我现在很怀疑这些时间到底花得值不值。

你询问 Erica 如何处理新的整合请求。她给你展示了客户面板上的一个很小的表格，上面显示："没看到你当地的分类广告商？请联系我们，我们会尽快支持它！"

Erica 解释道，团队通过这个表格收到了很多请求，但很难全部满足，因为每个服务集成所需要的时间大不相同。有时遇到的是基于 Web 的 API，很容易处理；但有时会遇到特别的请求，可能是特殊的邮件报告，或是上传到年代久远的 FTP 服务器上的电子表格，甚至是由传真机发来的文本文档。

TagSail 对分类广告网络的支持可谓不堪一击，但销售团队（不知何故）确信这一模式能为客户减少麻烦，而且肯定会有回报。你怀疑这里面有问题，于是开始深入调查。

7.3 注意权衡工作的经济效益

> 很多情况下，20% 的投入就会有 80% 的产出。
>
> ——帕累托法则

你用几分钟时间检查了一下项目的问题跟踪器，发现新请求的流入速度比现有请求的解决速度快 4 倍。如果算上 bug 报告，实际比例将近 8:1。

这个比例并不合理，因为它意味着大部分请求处于遥遥无期的等待状态，而且团队积压的工作越来越多。如果不加以解决，这在日后维护起来会比现在更令人头疼。

很明显，分类广告的整合过程有问题。但其严重性取决于分类广告能给公司带来多大效益。为了更全面地了解情况，你又问了几个问题。

你：广告网络整合的商业模式是怎样的？

Erica：除了针对发布售卖活动信息收取基本费用之外，我们还根据外部成本向客户收取广告费。

你：换句话说，整合本身并不产生直接收益，而只是为客户提供的一种福利？

Erica：没错。说实话，我们本来并不打算面向全国推行这项服务。一开始，我们只整合了新英格兰地区的一家主要的广告商，希望这样做能起到宣传作用，并且吸引更多的付费用户。我们确实做到了。但是，我们并没有下一步打算，而请求却源源不断地涌进来。

你：让我猜猜：早期成绩一定让销售团队很兴奋，然后他们顺水推舟，希望你们支持的广告商越多越好，对吗？

Erica：是的，而且他们根本没怎么和我们商量。首次整合是在一天内完成的，可以服务数十个城市。他们根本没想到，与服务这样小规模的市场相比，之后的整合所需要的时间要长得多。

你：我想我开始明白问题所在了。

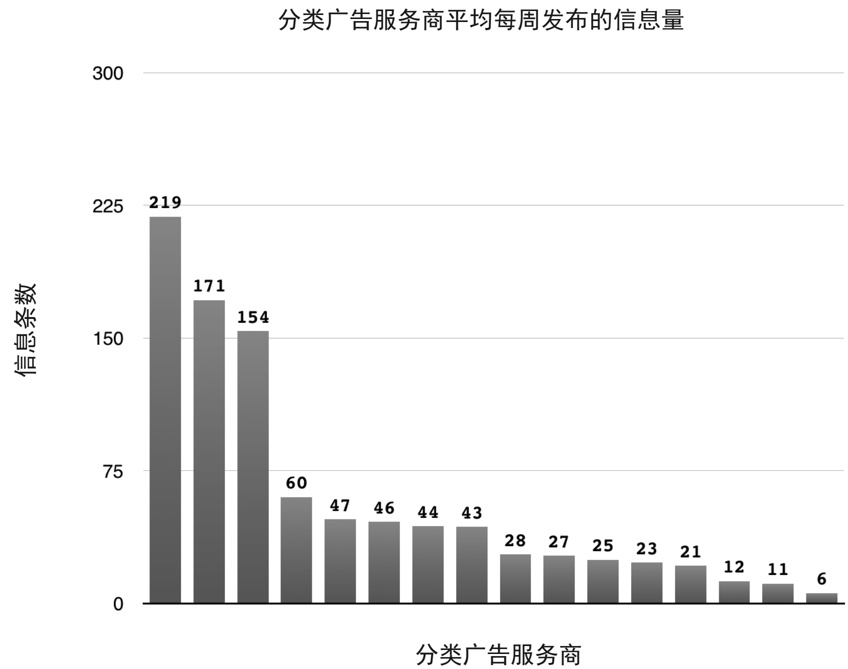
在 Erica 的帮助下，你做了一次小规模的市场调查，并且发现了一份统计报告。[2] 该报告显示，全美平均每周举行约 165 000 场二手物品出售活动，而最大的分类广告网站只列出了约 95 000 场。

Erica 快速查询了 TagSail 的数据，发现其每周约发布 15 000 条活动信息，还不到全国平均量的 10%。

在发布信息的客户中，大约一半使用至少一个分类广告服务。在这约一半的客户中，有 1/8 的客户选择通过支付额外费用使自己的信息登上当地的报纸和新闻网站。这样算起来，平均每周约有 1000 条广告使用了分类整合特性。

然后，你让 Erica 算出每个分类广告服务平均各自发布多少条信息。根据她算出的原始数据，你制作了如下图表。

注 2：虽然本故事是虚构的，但这份统计报告真实存在。如果你感兴趣，可以在 http://pbpbook.com/stats 找到这份报告。



虽然现在看来这样的数据不难理解，但想想还是不太舒服：近 3/5 的分类广告是由前 3 个服务商发布的，而后一半的服务商只发布了略多于 15% 的信息。本来使用分类整合特性的客户就相对较少，去支持这些不太流行的服务对于开发团队来说几乎完全是浪费时间。

你鼓励 Erica 把这些发现报告给产品团队的主管 Jen。她犹豫了，很担心自己的想法不会被重视，因为这和销售团队之前的想法背道而驰。但你指出，无论是产品团队还是销售团队，都没有接触过生成新报告的数据。于是，Erica 又重新燃起希望，自己这次的想法可能会被重视。

Jen 和 Erica 一起讨论了这个问题。你作为仲裁人旁听。大家都基本同意，大部分整合工作是在浪费时间。然后，你针对如何找回平衡提出了几个具体的建议。

- 主要支持 8 个最流行的服务商，它们发布了 83.6% 的信息。
- 为每个月用于处理整合工作的生产力设置固定的上限（比如刚开始可以设置为 20%）。如果团队的工作量超过了这一上限，就向产品团队进行反馈，以便后者视情况修改计划。
- 检查后 8 个服务商的情况，决定应该对它们进行何种级别的支持（如果真有必要支持）。可以留下维护开销较少的服务商，维护开销高的应该逐渐淘汰。

- 在整合更多的分类广告服务之前，与产品团队一起评估潜在市场的规模以及实现和维护分类整合特性的成本。确保花在整合工作上的时间不是从其他可能有价值的工作中挤出来的。
- 明确告知客户，不能保证支持新的分类广告服务商，并考虑彻底摈弃请求表格。
- 一旦整合工作量稳定下来，就实施预防性维护措施，如优化监控、日志记录、分析和测试等。根据经历过的痛点(而不是理论上的未来需求)，将这些预防性措施按优先级排序。

在开发应用时，有一部分工作的收益是递减的。上述计划的目的就是限制花在这部分工作上的精力。尽管经常被忽略，简单的时间安排仍能有力地限制项目中高风险部分的影响，同时鼓励更谨慎的优先级设置和成本效益分析。

即使只做到了计划中的一部分，开发团队也能轻松很多，同时省下大量时间去做更有效益的工作。

7.4　限制积压工作，力求减少浪费

4 周过去了，你又和 TagSail 团队碰面，看看他们工作得怎样了。你的第一个问题是，他们是否已经有效处理了分类广告整合问题。Erica 很愉快地告诉你，他们似乎终于使这一部分工作回到正轨了。

于是，你又问他们这段时间有没有新的进展。Erica 很遗憾地告诉你，并没有多少，除非把 bug 修复、杂活和内部代码清理都算上。在一阵沉默后，你们重新振作起来，开始分析原因。

> **你**：我不太明白。团队不是已经节省了约 30% 的生产力，还大力减少了意外的紧急工作吗?
>
> **Erica**：是的，但是产品团队一发现我们不用每天“灭火”了，就开始给我们堆积大量工作，想要充分利用省下来的生产力。
>
> 他们好像沉迷于“追赶”我们之前制订的产品计划，然而欲速则不达，我们又陷入了困境。
>
> **你**：究竟发生了什么呢?是不是匆忙做完工作但毫无质量可言，只是为了有时间应付下一批任务?
>
> **Erica**：不是，倒不是这种情况。实际的情况是每周都会计划并分配新工作，但产品团队动作很慢，拖着不回答我们的问题，也不为我们确认已完成的工作，这样我们就没法交付。我们自己的代码审查和 QA 测试也跟不上进度，因为整个团队接到的工作比完成的多。

你：不管你信不信，这其实是进步的标志。突破工作过程中的一个瓶颈，很自然地会让人看到另一个瓶颈。[3] 现在看来，新的约束是发布前的审查和批准速度。找到合适的节奏并坚持跟上进度可能需要耗费一些精力，但节奏一旦形成，你就能实实在在地看到自己在进步。

Erica：如果你的意思是让产品团队减少每周分配给我们的工作量，那绝对是行不通的。在进行新一轮融资时，他们就制订了计划，所以他们一直顶着巨大的压力，需要快速完成新的工作。

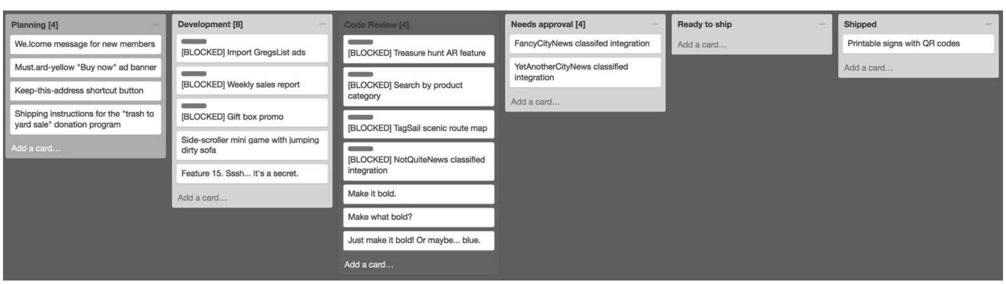
你：但是那些卡在代码审查环节的工作并没有完成，排着长队等待产品经理批准的工作也没有完成啊。你可能有 100 项处于积压状态的任务，但只有交付之后才能真正创造价值。

Erica：我同意，但怎么说服产品团队，让他们改变做法呢？

你：很简单，也很好办：需要让他们建立一种心态，即未上线的代码不是资产，而是存货。而且，这些存货还容易腐坏，并具有持有成本。

Erica 之前给产品团队解释过这一问题，但她是从其他角度切入的。她强调了切换工作内容的害处，以及未完成手头工作就开始新工作会产生的紧张情绪。

你指出，虽然这些担心都是完全合理的，但最好重新组织语言，重点强调产品团队目前的做法会使开发团队无法给客户带来新的价值。为了找到一些证据支持此观点，你让 Erica 给你看看开发团队的看板记录。



你立即发现，板子的右边（代表可交付的和已交付的工作）几乎是空的，而左边和中间部分（代表计划中的和正在进行的工作）被挤得满满当当。还有很多被阻塞的任务，如果将它们的阻塞状态同时解除，会造成极大的工作负担。

要解决这一问题，你需要和 Jen 聊一聊，因为她负责决定每周要做的工作。Erica 打通了她的电话并在一旁听着，你和 Jen 在电话里试着深入探讨项目计划速度和交付速度之间的巨大差异。

注 3：欲详细了解这个概念，请学习 Eli Goldratt 的“约束理论”（http://pbpbook.com/goldratt）。

你：几周前，我给团队做了一些调整，以为能够加快开发过程。但根据 Erica 告诉我的情况，好像事情进展得不是很顺利。你怎么看待这个问题？

Jen：有好有坏吧。好的方面是，开发人员好像最近不会被紧急的 bug 修复请求搞得那么紧张，所以他们现在可以集中精力处理效益更高的工作。但 CEO 和投资人还是给我们施加了不小的压力，让我们快速开发产品，而且我们的很多合作伙伴现在都摇摆不定。

你：开发团队面临的主要问题是，在他们还没来得及发布，甚至还没来得及完成手头任务的时候，新任务又来了，而且越堆越多。据我所知，他们的抱怨是合理的。

最近的一次面向客户的重大改进是在 6 周前完成的，现在有两个新特性等待确认。但是还有 12 个新特性处于正在开发的状态，外加 4 个正在计划之中！

Jen：我知道，情况很糟糕。但我们又能怎么办呢？

你：你们每周能够真正发布几个新特性呢？

Jen：我们原本计划每周进行一次重大改进和发布 2 ~ 3 个较小的特性。本来还计划每周二向客户群发邮件，告知新特性并展示如何使用。

你：那样做表面上看起来很合理，但是计划新特性的速度要远远高于发布速度。结果，你们不断积压正在进行中的工作，其中大部分都因为未得到反馈而被阻塞。重要的是总体流量，而你们的入流量远远大于出流量，这样就会过载。

Jen：我明白你的意思。这个问题的部分原因是，我们希望开发人员忙碌起来。也就是说，每当看板图上出现空位时，我们就认为应该坐下来和开发人员一起计划一个新特性了。现在他们的开发速度加快了，实际上这对于我们来说也耗费了不少时间。

你：为什么不利用这些时间为开发人员解答问题，或者确认他们的工作呢？这样不就能确保他们一直很有效率，而且可以更快地交付更多特性吗？

Jen：如果能自己做决定，我当然会那样做。但是开发人员提出的问题，很少有我能直接回答的。有些需要和销售团队沟通，有些需要进行客户调研，有些需要和供应商或合作伙伴商讨，有些甚至需要找 CEO 坐下来仔细探讨。

对于某些问题，只要找到对的人，10 分钟就能解决。但很遗憾，一般需要等一周甚至更久才能收到“对的人”的回复。

你：好的，现在我明白问题所在了。你们的反馈速度跟不上开发团队的进度。

> 若想跟上进度，要么放慢发布节奏，并且减少开发工作，要么加速反馈环。我认为将三者结合可以达到最佳效果。[4]

CEO 一开始拒绝这个想法。但在与他长谈之后，你和 Jen 终于设计出一个非常理想的方案。

- 暂停 4 周制订大的新特性计划，以便完成目前的工作。
- 调整发布周期：每两周发布一个主要特性。接下来的 4 周用于开发调整后的第一个主要特性。
- 滚动发布较小的特性，而不是通过公告阻塞相应的开发任务。
- 逐渐建立起一个包含 5 个可发布特性的储备库。在主要特性的开发任务被阻塞时，该储备库可当作缓冲。
- 一旦建立起储备库，便将开发计划与发布日程同步，使新发布的（或被取消的）特性触发新的主要特性的开发计划，而不是尽力用光开发力量。
- 复查产品计划，并将其减半。让销售团队和开发团队参与修订过程，以了解修订计划的成本与效益。
- 每周一上午抽出 4 小时，专门作为“协作时间”。在这段时间里，全体员工都不需要埋头工作或出席正式会议；相反，整段时间都用于互相解答问题，帮助被难点卡住的人。

这一方案的总体目标是给整个公司一些空间“松口气”。让待完成的任务在被新任务挤压之前就完成，应该会让公司的工作节奏更加自然和稳定。

这一方案仅仅是一个开始。你明确地告诉 Jen 和 Erica，在实施这个方案的过程中还会遇到困难，因为不能确保所有员工都按计划做。因此，你建议先试运行 12 周，看看效果如何。

7.5　力求整体大于部分之和

12 周的时光转瞬即逝。正如你所料，并不是每个人都愿意遵守规定。一些团队成员几次想要恢复过去的行事方式。他们虽然不情愿，但大体上还是遵守了规定，而且你的一些想法确实有效。

为了帮助你了解过去 12 周的进展，CEO 要求每个部门以“玫瑰、花蕾和荆棘”的形式总结这段时间的工作。

“玫瑰”代表发生的好事，“花蕾”代表大有希望的事，而“荆棘”代表遇到的痛点。

注 4：Donald Reinertsen 所著的 *The Principles of Product Development Flow* 是有关这一主题的绝佳读物（内容有些抽象）。

	玫瑰	花蕾	荆棘
销售团队 (Steve)	在过去12周里发布的5个主要特性都如期交付	持续交付较小的特性也许有利于提升产品声誉	目前的增长速度仍然远远低于预期
客户支持团队 (Lena)	新特性的平均缺陷率显著下降	客户开始发现我们能够快速响应紧急问题	因为优先级不够，所以许多小问题和不错的特性需求被拒绝处理
产品团队 (Jen)	未完成的任务减少，设计师更能关注质量，而非数量	包含可发布特性的储备库慢慢建立起来，我们能够更灵活地决定发布什么，以及何时发布	销售团队在做工作计划时仍然强调新要素，而没有专注于改进现有特性
开发团队 (Erica)	现在，工作很少因为未收到反馈而受阻；即使受阻，影响也极小	宽裕的时间有利于我们逐渐还清技术债	我们没能参与产品设计过程，这给编程工作带来了不必要的困难

尽管公司在过去经常跨部门召开会议，但是这可能是第一次从整体上展示决策对每个人的影响。各部门愿意做这件事，已经算是进步了，虽然很多问题还有待解决。

4 个部门主管谈及各自的“荆棘”时都不太自在。很快，你发现他们都各执一词。于是，你冷静地建议大家稍事休息。

会议再次开始后，你让他们各自写下一段简短的话，解释每个痛点的“痛处”。然后，你将他们写下的回复整理到一起并展示出来，这样大家就能从全局的角度看到各自关心的问题了。

	痛点	痛处
销售团队 (Steve)	目前的增长速度仍然远远低于预期	如果接下来6个月的总收益不能增加50%，公司将面临很大的财务问题。也许需要寻找新一轮投资或面临裁员
客户支持团队 (Lena)	因为优先级不够，所以许多小问题和不错的特性需求被拒绝处理	许多小问题在可容忍的范围内，但仍对客户有不良影响。正所谓“千里之堤，溃于蚁穴”，如果长期得不到解决，这些小问题终将成为大问题
产品团队 (Jen)	销售团队在做工作计划时仍然强调新要素，而没有专注于改进现有特性	目前的做法是一有新想法就融入产品设计。这让产品拥有令人印象深刻的诸多特性，但使产品设计过程支离破碎
开发团队 (Erica)	我们没能参与产品设计过程，这给编程工作带来了不必要的困难	产品团队直接给我们分配任务，而没有与我们商讨技术上的可行性。他们这样做是在不知晓开发成本的情况下猜测特性的价值

讨论继续进行，但这一次，你志愿从中调停，以保证大家不跑题。

> **你**：我打算从 Steve 关注的问题开始，因为这是最大的问题。产品的确可以赚钱，但是现金流不是正向的，更不用说盈利了。公司要养活 20 多个员工，因此这个问题尤其可怕，需要每个人都警惕起来。
>
> Jen：这是我第一次看到 Steve 从资金周转角度解释这一问题，而不是展示收益增长情况。我认为公司里的每个人都能理解前者，后者好像比较抽象。
>
> Erica：假设我告诉开发团队"如果我们没办法快速扭转局面，6 个月之内，你们之中的某些人就会丢掉工作"，我很难想象他们会作何反应。
>
> Lena：我同意，而且我确定，如果需要裁员，客户支持团队首当其冲。这真是坏消息啊。
>
> Steve：很不幸，确定合理的增长率目标这件事，销售团队做不了主；我们只能接受 CEO 和公司投资人下达的任务，努力达到他们给我们定的目标。他们要求的增长曲线比我们实际能达到的要陡得多，而且这种情况已经持续好几个月了。
>
> **你**：但这样不就意味着，产品计划也要基于这种不合理的增长曲线吗？换句话说，现在的策略是不是"要么做大，要么回家"，但我们根本没有那么多资源做大？
>
> Steve：嗯，我认为可以这样说。究其原因，要维持公司的运营，接下来 6 个月需要达到至少 50% 的收益增长率。要想让投资人高兴，接下来的 180 天需要达到大约 150% 的收益增长率。
>
> **你**：为了便于讨论，我们假设公司没有办法实现这样的增长目标。如果把目标减半，销售团队能不能将工作重点转移到已完成的特性上一段时间，而不是把赌注押在还未开发的主要特性上呢？
>
> Steve：这个需要请 CEO 批准，但可以试一两个月。而且我们还得证明这种方法一定会带来收益才行。

你向房间里的所有非销售人员指出，在一个还未盈利的公司里，现金即是氧气——没有了它，一切很快就会变糟。思考这种事并不令人愉快，但如果不保持警惕，就可能面临失业。

同时，产品的经济效益与员工的团队合作能力直接相关。要开发更优秀、更有凝聚力的产品，需要平衡产品开发过程中所有相关人员的需求，不能偏袒某一团队而忽略其他团队。

综合考虑所有“玫瑰、花蕾和荆棘”之后，你为团队制订了一个计划，帮助他们在接下来几个月的重要工作中保持紧密联系。

- 创建一个新的面板，列出核心的海盗指标[5]（AARRR metrics，AARRR 分别代表 Acquisition、Activation、Retention、Revenue 和 Referral，即获取、激活、留存、收入和推荐），这些指标对任何公司都至关重要。让每个人看到同一份报告，并训练所有员工学会阅读报告，以便让所有人迅速了解产品的总体健康状况。
- 找到 AARRR 管道的瓶颈，然后让所有团队共同尝试解决难题。从渐进式改进开始，逐步扩展到更实质的改变。
- 开一次全体会议，回顾产品的使用情况，对象既要包括寻找活动信息的用户，也要包括发布活动信息的客户。让每位员工都记下任何可以改进的地方。
- 查看出现的问题中有没有与已有支持请求或产品计划中的问题重合的内容。将重复问题设定为近期内需要优先解决的问题，然后用海盗指标估算其影响。
- 每周让一位开发人员抽出一天时间解决客户支持团队认为值得处理的“小问题”。每周轮换，让所有开发人员都有机会处理客户的问题，从中获得经验。
- 尽可能多地安排交叉培训的机会。开发人员和产品设计师应当参与一些销售方面的活动，销售人员也应该参加一些项目计划方面的会议。另外，让公司里的每个人每月都至少花一小时参与一线客户支持工作。
- 在接下来的 8 周里，找出 3 个可以被移除或可以大大简化的产品特性。尤其注意那些与整体的产品系统没有紧密结合的特性。

上述措施有一个共同的原则：要足够了解周围每个人都在做什么，以便了解自己的工作是否符合大局。

讨论过这一计划后，各个团队对接下来几个月的工作表现出了乐观情绪。虽然并不能保证一定成功，但比起前几天，他们之间的凝聚力增强了许多。

你用夸张的语气说道：“我的使命完成了。”然后，你便策马扬鞭，逐渐消失在落日的余晖中。而对于其他人来说，工作才刚刚开始。

注 5：http://pbpbook.com/aarrr

忠告与提醒

- 在处理系统级故障时，要根据需要停用或降级一些特性，使软件尽快回到可用状态。一旦压力得以缓解，便可以开始认真修复发生故障的部分。
- 寻找过度耗费时间的部分，用合理的安排加以限制，这样就可以省下时间进行其他工作。做决策不能只靠直觉，要使用“毛估”计算法，将经济因素也考虑进去。
- 谨记未上线的代码不是资产，而是存货。存货容易腐坏，并且具有持有成本。要帮助项目中的每个工作人员理解这一点。做法是让他们集中精力在给定的时间段内交付有价值的工作，而不是让团队中的每个人都为了忙碌而忙碌。
- 与分工不同的人合作时，试着用对方能理解的方式进行交流。从旁观者的角度看待问题，并思考：“这件事与正在和我交谈的人有关吗？这件事对整个项目有何影响？”

问题与练习

问题 1：找出你在当前的开发过程中遇到的“玫瑰、花蕾和荆棘”，各列一项。你的开发同事列出的内容与之相似还是不同？项目中的非技术人员又如何呢？

问题 2：思考你用来衡量项目总体健康状况的指标。它们能否精准地体现有意义的事实？如果不能，原因是什么？如果能，那么再过 6 个月，这些指标还有效吗？到时是否需要采用其他指标？

练习 1：选择一个你正在维护的项目，给项目的某一特性引入一个重大故障，然后将其部署在测试环境中。在不“修复”受损特性的情况下，想办法通过应急措施或隐藏问题，在 15 分钟之内恢复尽可能多的系统功能。

练习 2：连续 4 周记录工作日志，列出每天的主要活动。找出你认为相对于所花时间，可能获得最高收益的 3 个项目或周期性任务。是什么让这些工作脱颖而出？有没有指标可以衡量你的成果？

第 8 章

认清行业未来，再议软件开发

和在前几章中一样，你在本章中也会读到一个小故事。但由于想把本章作为后记，因此我想暂停自己的旁白身份，转而说一些重要的事。

我之所以撰写本书，是因为我相信程序员不只是编程专家，转型在所难免。如果这个观点正确，那么整个行业都应做好准备，因为在不久的将来，“程序员是用技术解决人类社会常见问题的人”这一观点可能成为行业标准。

我已经写了几十年代码，感觉这个想法有些激进，但也令我如释重负。对于我来说，程序设计中最有趣的一直是解决问题、沟通等“以人为本”的方面；代码只是我能找到的解决问题最有力的工具而已。

本书中的故事不包含示例代码，但它们的目标很明确：帮助你我关注软件开发中很多有趣的、更高层级的挑战。但在每个故事的背后，都有大量的代码编写工作，只不过我们的关注点不在代码上而已。

现在让我们更进一步，想象在未来世界中，机器会完成大部分代码编写工作。我承诺在故事结束时，一定会回到现实中，但我们不妨在大幕落下之前找些乐趣。

本章主要内容

你将会初步了解，当把精力完全放在解决问题和沟通上，而不管代码怎么写时，程序设计会变成什么样。

想象你正处于一个几乎空着的房间中央。过去 5 年里，这里一直是你的办公室。但直到现在，你仍然会在每次走进这个房间时有一种置身于老科幻电影之中的感觉。

要使用这一工作场所，你只需戴上一副特制的护目镜和一副手套；埋在墙里的大量传感器、摄像头、扬声器和其他电子器件会安排好一切。

透过护目镜，房间变成了一间可爱的小办公室。它在高高的大楼里，城市风光尽收眼底。这背景很好，但工作才是重点。

你叫出了虚拟助手 Robo，向它示意你已经准备好开始今天的工作了。Robo 马上响应，并贴心地提醒你工作内容。

> Robo：您好！未来城交通部的 Carol 让您准备一份报告，帮助她给今年的人行道维修工程设置预算。我是不是应该加载项目笔记，好让我们开始工作？
>
> 你：是的。我们先来看看城市服务请求公共数据吧。请给我展示一下未来城 311 号 API 的服务代码表。
>
> Robo：这张表是一组很简单的键值对。如果需要查看全部内容，请看正前方的窗口。如果需要查找某个特定键，请告诉我，我会为您突出显示。
>
> 你：我在找人行道。
>
> Robo：我找到了与“人行道和路缘损坏”匹配的键，值为 117。
>
> 你：没错，就是这个！

你将身子向前探了探，在空气中捏了一下，捏的位置就在你透过护目镜看到的“人行道和路缘损坏”这一行。捏过之后，这一行在护目镜中变成了一张小即时贴。你将它放到你的（虚拟）口袋里。然后，你对着正前方的巨大数据表拍了一下，它便立即消失了。

你让 Robo 展示未来城 311 号服务提供的所有数据源的概览情况。办公室左面的墙马上被文档占满了。你抓住有问题报告表的一页，然后坐下来开始思考。

之所以需要制作这份报告，是因为未来城想要算出维修人行道所需要的费用，并想了解费用应该花在城市的哪些地方。作为开始，计算供给量和需求量可能会有帮助，这也是你查看 311 号公共数据的原因。[1]

现在还不知道究竟要寻找什么，但你决定先进行几个可视化操作，看看情况如何。

注 1：事实上，城市数据现在已经对公众开放了。本章的例子正是基于 SeeClickFix 的数据，详见 http://dev.seeclickfix.com。

你：Robo，我们先根据问题报告表做一些工作吧。我现在手里有数据表，你可以用它进行查询。

Robo：好的。您想让我对该数据源进行什么操作？

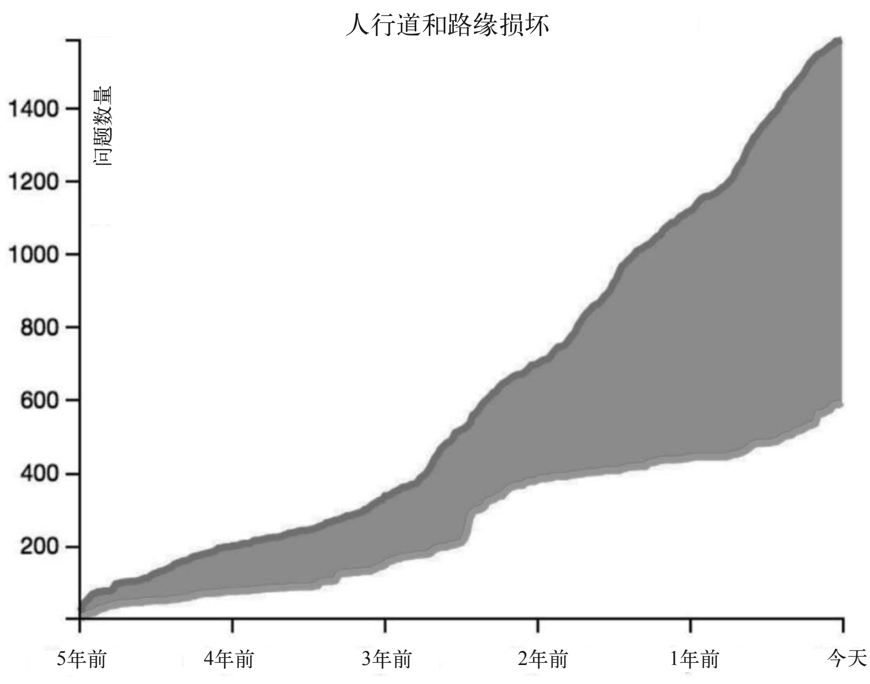
你：数据表包含收到问题和解决问题的时间戳字段。请根据这些数据绘制一张过去 5 年的累积流图，以周为分割单位。请做出图和数值表两个版本。

Robo：完成！请查看显示的内容是否符合要求。

你：坏了，我忘了些东西！ Robo，请只显示“服务代码”字段与便签条上的数字相匹配的记录。

Robo：哔哔哔！完成。

你看了一眼数值表，检查是否正常。内容如你所料。然后，你开始看图。



在过去几年里，市民的投诉量猛增。从图中很容易看出为什么为未来城设置预算这么难：未来城根本没有足够的资源彻底解决这一问题，所以需要精打细算，充分利用手头有限的资源。

你让 Robo 显示调研文件，办公室右面的墙随即出现数十页资料。其中包括你上次为未来城工作时搜集的材料，Carol 第一次要求处理这个项目时给 Robo 分享的文档，还有 Robo 用内置的推荐引擎找到的额外资源。

你：非常感谢这一墙的字，Robo。请用一段话总结一下这些文字。

Robo：未来城的人行道年度预算是按照片区划分的。如果一个片区经常需要维修人行道，并且其 PCI 值[2] 为 20 ～ 60，那么该片区就拥有最高优先级。PCI 值低于 20，表示维修成本大大高于当前的操作预算。PCI 值高于 60，表示道路足够耐用，可以推迟维修。

你：谢谢。请突出显示所有提到 PCI 或片区的文档。

Robo：哔！完成。

Robo 一直都很聪明，它相当于一个强大的搜索引擎和一个结构合理的知识库的结合体。但思考工作还是要人类来完成，所以此时工作才刚刚开始。

你花几分钟仔细查看了突出显示的文档，想寻找一些灵感。其中一个文档包含了对未来城数百个地点的路面状况的详细评估，该文档是在去年由未来城工程部发布的。你像摘水果一样把这个文档摘取下来，扔到了你常用的主工作区中。文档“啪”的一声清晰地显示在墙上。

然后，你询问该项目的调研文件是否包含未来城的地理空间数据，Robo 明确地回答说没有找到匹配的数据。你又重新组织语言，询问能不能在互联网上找到未来城片区划分的相关地理空间数据。Robo 找到的第一条搜索结果正是你想要的那组 shape 文件。

工程部发布的那个文档包含街道地址及其人行道的 PCI 值。你让 Robo 先将街道地址转换为地理坐标，然后把这些地理坐标和未来城片区的 shape 文件对应起来。这样就生成了新的数据表，并且每一行都有了片区名称信息。

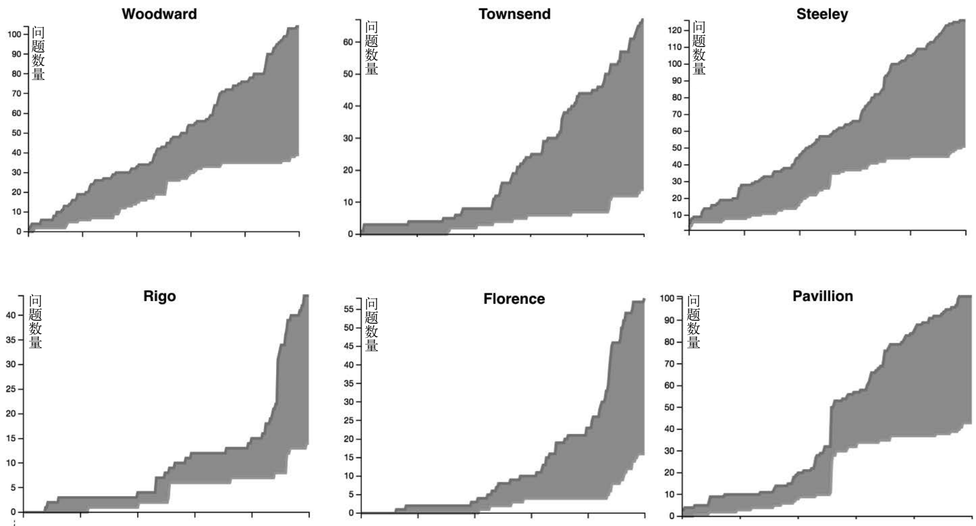
一生成数据表，你便做了统计，根据 PCI 值为 20 ～ 60 的样本比例对每个片区进行评级。这部分工作可能需要市政府工作人员审核和调整，但你算是为他们提供了一个很好的切入点。

最后，你回到之前绘制的累积流图，指导 Robo 根据片区给问题分组。你不想再口头重复，于是切换到可视脚本模式，复制相关的逻辑块，将街道地址映射到片区，然后在人行道问题报告中的位置字段也应用了相同的操作。

稍作修改之后，你最终绘制出人行道维修供需历史图，以片区划分，并按照路面状况评级排序。你将前 6 个片区放在工作区主墙上，其余的先挂在一边。

你给 Carol 发了一条消息，告诉她报告已经做好，可以查看了。几分钟之后，Carol（其实是她的影像）已经站在了你身边，她能看到你正在查看的内容。

注 2：PCI 即 Pavement Condition Index，路面状况指数。PCI 的取值范围是 0 ～ 100，值越大，说明路面状况越好。——编者注



在你简单解释报告背后的逻辑时，Carol 走来走去，用手指捏了图中的好几个区域，在不同的时间点上放大。同时，Robo 录下了你们谈话的视频，并自动将对话内容转换成文字，供以后参考。

Carol：这个切入点非常有帮助。我认为下一步是查看这些评级较高的片区，然后在街区级别上再深入研究一下。

移动人行道维修设备的成本很高，所以维修地点离得越近，批量维修的成本就越低。理想的情况是需要维修的地点全部集中在一条街上，这样就根本不需要移动维修设备。

你：好的，我可以做些工作来解决这个问题。工程部应该有人能给我提供一些准确数字，用来根据维修地点间的距离进行加权，对吧？

Carol：我觉得可以，我会让他们给 Robo 传一些文档。另外，就在今早我和一个朋友经过“大未来城”时，听他们说想和我们分享自己的调研文件，只要我们同意以后把我们的文件也分享给他们。你也许可以浏览一下这些文件，看看其中是否有内容可以用于我们的报告。

你：我会的。还有什么是我可以帮你的吗？

Carol：哦，我本不喜欢提到这件事，但还是要问一下。到目前为止，你做这份报告算多长时间的费用？我们比较在乎费用。

你：一小时多一点吧。我认为需要继续做街区分析，可能还要深入研究“大未来城”的调研文件，中午之前应该可以完成，大约总共花 3 小时。

Carol：太好了，谢谢！

Carol 的影像逐渐消失在空气中。在她离开一会儿后，你的虚拟工作区也渐渐消失。你还是原来的你……在你面前有一本编程书。有着奇怪幽默感的作者，正用他那独特的方式向你道别。

* * *

你刚刚读到的这个故事描述了我梦想的未来程序设计的样子。那时我们已经跳出编程语言的桎梏，不用再自己动手编写代码了。

本章所描述的虚拟工作流只是可能在未来出现的一种人机界面，它满足我真正的需求：站在足够高的层次与计算机交流，只需要关注如何解决问题，而不是纠结于代码编写中的细枝末节。

本书从头到尾讨论了突破工具限制的许多方法，但我完全承认，书中的建议只能在一定程度上减少我们每天需要处理的问题。为了使计算机行业发挥所有潜能，我们需要一种开发工具链，其设计应完全以人为本、为人类服务。

在今天已取得的成果中，其实有很多值得借鉴之处，尽管可能只是在某些具体问题上有效。我们已经习以为常的电子表格就是一个很好的例子，我也是从电子表格中受到启发，才编写出本故事中的交互行为。

- 如果想求电子表格中一列数字之和，只需要点击几下鼠标和按几下键盘。心里想着“我要把这些值加起来”，然后选定数字，再在键盘上敲出 SUM 一词，数字之和就会出现。
- 如果想绘制时间序列图，只需选定一列时间戳和一列对应值，然后点击一个类似于时间序列图的图标即可。
- 如果想将工作成果分享给其他人，只需将文件发给他们或为他们提供共享文件的访问权限即可。

电子表格中的数据和处理数据的函数密不可分，源代码与程序也同根同源——它们只是一些文档，但功能强大且听命于你，不需要你把自己的想法笨拙地翻译成低级语言。

现代 Web 浏览器的开发者工具也给人类似的感觉。在其中，你可以直接和所在的页面交互，这将“文档”的概念与对象模型无缝衔接。

在文档中找到标签 `<h1>` 并改变其样式，这对于任何拥有基本 HTML 技能的人来说都不是什么难事。但这一操作浪费时间、偏离重点，让人觉得像是在写代码。相反，直接在 Web 浏览器中点击标题，立即进入内联编辑器，在其中查看元素的属性并实时修改，这会给人完全不同的感受。你可以在心里想着“我想增大这个标题的字号”，然后直接操作，而不是一边在头脑中模拟文档对象模型一边编辑静态文本文件。

那么，我们应该怎样在本章描述的人行道报告问题中应用相似的交互模式呢？答案很简

单——我们做不到，至少用现在的开发工具无法实现。所幸的是，在我们面前并没有什么硬性的技术限制，只是工具的使用习惯和现有设计阻碍了我们前进的步伐。

＊ ＊ ＊

作为思维实验，让我们快速回顾一下上文描述的人行道报告，并将所有虚拟现实和人工智能的元素抛开不管。以下是这类分析涉及的几个通用操作。

- 将街道地址转换为地理坐标。
- 将地理坐标与包含它的区域相匹配。
- 从 Web 服务导出数据表。
- 根据数据表，生成累积流图。
- 进行基本的数值运算（排序、求和、求平均值等）。

理论上，一套像样的工具应该在执行上述操作时不费吹灰之力，就像我们在电子表格里运算或使用浏览器的开发者工具更改样式一样。而在实践中，情况比想象的要复杂得多。

因为我制作过像故事中那样的报告，所以我知道，现实中有很多工具可以帮助处理上述每一个任务。即使你决定自己开发工具，也不会难如登天。

但是在使用现有的工具制作此类报告时，工作流程足够自然顺畅吗？当然不！当前的开发过程十分繁琐，需要考虑怎么把第三方库和所用语言的核心函数结合，怎么让数据格式和 Web 服务协议对应，等等。最后出来的产品像是用鞋带和胶水胡乱拼接在一起的，除非你愿意仔细研究代码，把它写“优雅”。但这样做成本就会太高，效益也不一定好。

由于存在这样的复杂性，因此在真正制作这份简单的报告时，如果你想保持头脑清醒、不忘记问题背景，就需要一遍又一遍地问：“等等，我想解决什么问题来着？”

＊ ＊ ＊

虽然软件开发行业还有很长的路要走，但是我认为未来几年会变得更好。的确，有一些开发人员只对工具、代码及其带来的智力挑战感兴趣。但对于其他人来说，这些只属于工作环境，而不是我们的本性。

我的基本观点是：程序员和其他人一样关心人类利益；只不过，他们很难将其作为生活重点，因为每天大部分时间都被花在查找缺失的分号、阅读没有文档的库的源代码，或是盯着某些可能因 Unicode 转换错误而损坏的二进制转储文件发呆。

我最大的心愿是，如果能够与粗糙、低级、繁琐的现有工具作斗争，逐渐用与工作成果更贴近的工具取代它们，那么行业的关注点就能果断地、永久地由以技术为中心转变为以人为本。

为了全面看待这一问题，可以想象一下在过去代码由汇编语言编写、数据需要存储在手动打包的二进制大对象里的那种环境下，该怎样解决现在的编程问题。在数字和逻辑约束的海洋里，很容易将整个软件开发问题看作纯数学问题。

这种抽象空间不太容易理解。因此，勤恳的程序员不愿被吹毛求疵，尤其是当他们觉得自己的工作很高端时更是如此。

现在思考一下：下一代程序员会怎样看待我们？

现在就行动起来，你将会有一个值得骄傲的明天。对于我们每个人来说，问题的答案不尽相同，但考虑这个问题是很有必要的，因为未来需要每个人参与创造、互相帮助、贡献力量。

我所贡献的，就是本书。我希望你喜欢它，也希望它能够给你一些启发、伴你前行。

感谢阅读，祝你工作顺利！



如果你在阅读本书的过程中有任何疑问，或是有其他想要讨论的问题，欢迎给我发邮件（gregory@practicingdeveloper.com），或通过 Twitter 联系我（@practicingdev）。

干得好！你已经读完了本书。

请享受最后一个挑战，将它作为临别礼物。[3]

$	20	:	A	$	25	:	B	>	0B	>	37	50	52	4F	47
52	41	4D	4D	49	4E	47	20	42	45	59	4F	4E	44	20	50
52	41	43	54	49	43	45	53	50	52	4F	47	52	41	4D	4D
:	X	$	A	:	Y	^	>	35	49	4E	47	20	42	45	59
4F	4E	44	20	50	52	41	43	54	49	43	45	53	50	52	4F
47	52	41	?	Y	AA	+	X	-	Y	>	35	4D	4D	35	49
4E	47	20	42	45	59	4F	4E	44	20	50	52	41	43	54	49
43	45	53	?	B	F9	-	B	>	03	50	52	4F	47	52	41
35	4D	4D	35	49	4E	47	20	42	4F	4E	44	20	50	52	41
35	4D	4D	35	49	4E	47	20	42	4F	4E	44	20	4F	4E	!
35	4D	4D	35	49	4E	47	20	42	4F	#	X	^	>	37	41
$	0E	$	48	$	53	$	49	$	46	$	00	$	45	$	48
$	54	$	00	$	4C	$	4C	$	41	$	00	$	52	$	4F
$	46	$	00	$	53	$	4B	$	4E	$	41	$	48	$	54
$	00	$	44	$	4E	$	41	$	00	$	0C	$	47	$	4E
$	4F	$	4C	$	00	$	4F	$	33	>	A0	50	52	41	43

生活是个谜，就像一个巨大的“该怎样”（what if），但生活的乐趣在于我们为自己设定（set）要选择的方向。我们有得（gain）也有失（lose）。但随着时间一天天流逝，我们会一点点成长。

注 3：友情提示：这道题目比前两道要难。但如果你能找出其中的“函数”，然后将其翻译为伪代码，就能够用纸笔解开这道题。手动遍历每个操作会很繁琐，因此你只需做一些力所能及的工作，验证一下设想即可。写程序来解码其中的信息会更有效率，但如果你能手动将低级操作逐渐分解为高级函数，会学到更多。若有兴趣，也可以两种方法都试试。:-)

作者介绍

Gregory T. Brown 自 2010 年开始独立出版期刊 *Practicing Ruby*。他是非常流行的 PDF 生成库 Prawn PDF 的原作者。

在咨询项目中，Gregory 与大大小小的公司合作过。他与这些公司的干系人一起确定核心的业务问题，并力求以最少的代码解决问题。

Gregory 发现，90% 的程序设计工作都不需要写代码。他竭力关注这部分工作，这也启发了他撰写本书。

封面介绍

本书封面上的动物是秘鲁蜘蛛猴（Ateles chamek），也叫黑脸蜘蛛猴。虽然名字中带有“秘鲁”，但这种灵长类动物也见于巴西和玻利维亚。它们生活在低地森林里，用长长的四肢和易于盘卷的尾巴极度敏捷地在高空的树枝间摆荡游走。

秘鲁蜘蛛猴身形修长，皮毛和脸都是乌黑的。雄性和雌性大小基本相同，平均体重为 6.8 ～ 9.1 千克，平均身长为 0.6 米（不包括尾巴，尾巴长度可达 0.9 米）。它们的身体极适合在树梢生活：长长的手指，极灵活的肩关节，以及部分无毛的尾巴尖（可以把树枝抓得更牢）。蜘蛛猴多以水果为食，同时佐以树叶、昆虫、鸟蛋、蜂蜜及鸟类或蛙类等小型动物。

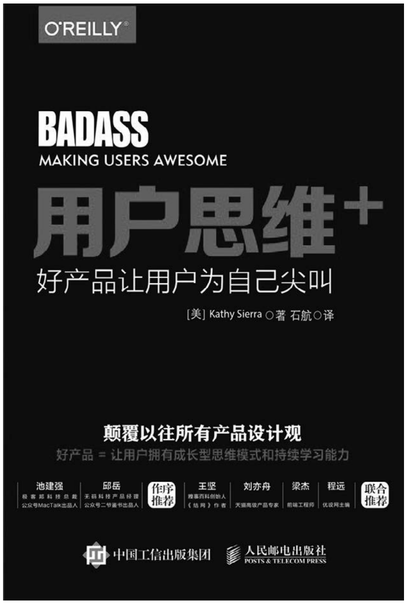
和大多数灵长类动物一样，秘鲁蜘蛛猴生活在社会群体中。群体的大小可随季节变化，因为雌性会离开群体生育，并于几个月后返回。新生的蜘蛛猴大约在 10 个月大时便可独立生活，但 4 年后才会性成熟。蜘蛛猴口头交流丰富，形式有尖叫、吠叫及马啸等，它们也会通过摇摆手臂和摇晃树枝来交流。

和许多雨林生物一样，秘鲁蜘蛛猴已经濒临灭绝。由于伐木和农业活动，蜘蛛猴的栖息环境越来越小；而且由于亚马孙地区的野生动物贸易，蜘蛛猴及其他大型动物被过度捕杀。

O’Reilly 图书封面上的许多动物都已濒临灭绝，它们都是自然界所剩无几的瑰宝。要了解如何帮助它们，请访问 http://animals.oreilly.com。

封面图片为版画，来源未知。

技术改变世界 · 阅读塑造人生



用户思维 +：好产品让用户为自己尖叫

- 颠覆以往所有产品设计观
- 好产品 = 让用户拥有成长型思维模式和持续学习能力
- 极客邦科技总裁池建强、公众号二爷鉴书出品人邱岳作序推荐
- 《结网》作者王坚、《谷歌和亚马逊如何做产品》译者刘亦舟、前端工程师梁杰、优设网主编程远联合推荐

书号：978-7-115-45742-4
定价：69.00 元



进化：从孤胆极客到高效团队

- 程序员版《人性的弱点》
- 提升职业生涯软技能
- 探讨领导力、合作、沟通、高效等团队成功关键因素

书号：978-7-115-43418-0
定价：45.00 元

O'REILLY®

代码之外的功夫：程序员精进之路

如果你认为程序员的工作就是写代码，那就大错特错了！事实上，编程只占程序设计工作的10%，而且是相对简单的工作。本书以8个虚构的故事为主线，邀你探索更有趣、更具挑战性的那90%的程序设计工作，构建行之有效的思维框架，从而提升解决问题的综合能力。

- 善用设计原型，探索项目创意
- 观察增量变更，发掘隐藏依赖
- 准确识别痛点，高效集成服务
- 设计严密方案，逐步解决问题
- 谨记自底向上，优化软件设计
- 认清现实瑕疵，改善数据建模
- 逐渐改善流程，合理安排时间
- 认清行业未来，再议软件开发

Gregory T. Brown，期刊*Practicing Ruby*出版人；非常流行的PDF生成库Prawn PDF的原作者；IT咨询顾问，帮助过各种规模的公司确定核心业务问题，力求以最少的代码解决问题。

“这是一本特立独行的书，篇幅很短，内容却很丰富，深入探究了一些软件开发中十分重要却少有人关注的问题。”

——Michael Feathers
面向对象技术专家，测试框架
CppUnit和FitCpp开发人员，
《修改代码的艺术》作者

PROGRAMMING

封面设计：Karen Montgomery　马冬燕

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机／软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-47837-5

ISBN 978-7-115-47837-5
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks